SYSTEM DESIGN PRINCIPLES FOR HETEROGENEOUS RESOURCE MANAGEMENT AND SCHEDULING IN ACCELERATOR-BASED SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Dipanjan Sengupta

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2016

SYSTEM DESIGN PRINCIPLES FOR HETEROGENEOUS RESOURCE MANAGEMENT AND SCHEDULING IN ACCELERATOR-BASED SYSTEMS

Approved by:

Dr. Matthew Wolf, Committee Chair
College of Computing
Georgia Institute of Technology

Prof. Karsten Schwan, Advisor
College of Computing
Georgia Institute of Technology

Dr. Ada Gavrilovska
College of Computing
Georgia Institute of Technology

Prof. Ling Liu
College of Computing
Georgia Institute of Technology

Prof. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Theodore L. Willke
Senior Principal Engineer
Intel Labs, Hillsboro, OR

Date Approved: 16 May 2016

To my parents.

ACKNOWLEDGEMENTS

First and foremost I would like to express my special appreciation and thanks to my advisor Prof. Karsten Schwan. It has been an honor to be his Ph.D. student. He has inspired me greatly to work in high performance computing (HPC) field. His willingness to motivate me and to help me grow as a researcher has contributed greatly to this thesis work. I am absolutely indebted to all his contributions of time, ideas, and research directions making my Ph.D. experience the most intellectually stimulating and enjoyable time in my career so far. I would also like to thank Dr. Matthew Wolf, Dr. Ada Gavrilovska and Dr. Jeffrey Young for mentoring and advising me at various points in my research.

I would like to thank the other members of my dissertation committee, Prof. Sudhakar Yalamanchili, Prof. Ling Liu and Dr. Theodore L. Willke for serving as my committee members and for their insightful comments and suggestions on my research. My mentors during my internships at Intel Labs, Xia Zhu (Ivy) and Narayanan Sundaram, have had a major impact on shaping the ideas presented in this thesis. I would also like to express my special thanks to my manager at Intel Labs Theodore L. Willke, who is still a mentor to me. Susie McClain, our group admin, who has been extremely helpful in taking care of our travels, day to day lab needs, etc deserves a special approbation. Not many people can keep that amount of organization and detail in their head.

My time at Georgia Tech was made enjoyable in large part due to many friends and groups that became a part of my life. I am grateful for the time spent and several intellectually stimulating discussions with my past and present labmates in the CERCS group over the years. A special mention to Minsung Jang, Ketan Bhardwaj, Bharath Srinivasan, Mukil Kesavan, Abhinav Narain, Sudarsun Kannan, Hrishikesh Amur, Vishal Gupta, Subramanya R. Dulloor, Min Lee, Fang Zheng, Hobin Yoon, Alex Merritt, Junwei Li and Jai Dayal. Mukil and Minsung were particularly very helpful in giving me general advice for a successful Ph.D. life. I hope to keep in touch with all of them in future. I am extremely

grateful to Sudarsun for being the nicest roommate ever. I have learnt so much from him and he always has the most sound advice I can get on life issues. I would also like to thank all my friends from IIT Kharagpur: Subhra Mazumdar, Puneet Jain, Satya Gautam, Pranith Kumar, Karan Doshi and Anchal Nema for helping me feel lighthearted during the most stressful moments in my Ph.D. life.

Finally, I would like to thank my family for their love and support, without whom I would have given up a long time ago. For my parents who have always supported me in all my pursuit be it my interest in higher studies or my love for liberal arts. For the presence of my elder brother one phone call away with his intellectually and spiritually stimulating advice. I would like to dedicate this thesis to my parents for what I am today is all because of them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Accelerator-based systems are making rapid inroads into becoming platforms of choice for both high-end cloud services and processing applications with irregular access pattern such as real-world graph analytics, due to their high scalability and low dollar to FLOPS ratios. Yet GPUs are not first class schedulable entities causing substantial hardware resource underutilization, including their computational and data movement engines. Therefore, software solutions with support for efficient resource management principles are required to address such scheduling challenges in GPUs. Further, two important characteristics of real world graphs like those in social networks are that they are big and are constantly evolving over time. This poses challenge due to limitations in GPU-resident memory for storing these large graphs. And because of the high rate at which these large-scale graphs evolve, it is undesirable and computationally infeasible to repeatedly run static graph analytics on a sequence of versions, or snapshots, of the evolving graph. Therefore, novel incremental solutions are required to process large-scale evolving graphs in near real-time using GPUs with memory footprint exceeding the device's internal memory capacity.

First, to address the challenges of GPU multi-tenancy, the thesis presents *Strings* scheduler for heterogeneous manycore nodes that implements a model in which GPUs are treated as first class schedulable entities, by decomposing the scheduling problem into a combination of load balancing and per-device resource sharing. Its utility as an infrastructure for developing and evaluating advanced scheduling methods is demonstrated for server workloads, where (i) load balancing intelligently binds each applications GPU component to an appropriate GPU and, (ii) device-level sharing aims to keep all of a GPUs hardware units busy, by concurrently running those applications that reside in different phases of their use of the GPU. It also prioritizes GPU requests that have attained 'least service' to achieve high system throughput, and goes beyond that to also ensure fairness via a history based fair-share scheduler. Over a wide variety of multi-tenant workloads, Strings achieves

substantial speedups compared to that obtained by the native CUDA runtime and other competitive GPU schedulers.

Second, to address the problem of processing graph applications with larger memory footprint than the device memory, the thesis presents *GraphReduce*, a highly efficient and scalable GPU-based framework that adopts a combination of edge- and vertex-centric implementations of the Gather-Apply-Scatter programming model and operates on multiple asynchronous GPU streams to fully exploit the high degrees of parallelism in GPUs supporting efficient graph data movement between the host and device. GraphReduce (GR) runs graph algorithms on GPUs without unduly burdening graph algorithm developers. Programmers write the appropriate sequential codes for their graph algorithms and then use GR's simple APIs to express their use for processing various graphs. The GR runtime seamlessly partitions the graph into different shards, each single one of which entirely fits into GPU memory, and overlaps shard movement with GPU-level graph processing, the latter using multiple levels of GPU-level parallelism. With such automation, GR can deal with graph sizes much exceeding GPU memory sizes. Extensive experimental evaluations for a wide variety of graph inputs and algorithms demonstrate that GraphReduce significantly outperforms other competing out-of-core approaches.

Finally, we address our original motivating problem of processing real-world graphs that are constantly evolving over time using GPUs. Although modern GPUs provide massive amount of parallelism for efficient graph processing, the challenges remain due to their lack of support for this near real-time streaming nature of dynamic graphs. Specifically, because of the current high volume and velocity of graph data combined with the complexity of user queries, traditional processing methods by first storing the updates and then repeatedly running static graph analytics on a sequence of versions or snapshots are deemed undesirable and computationally infeasible on GPU. To address this problem of analyzing evolving graphs in near real- time, we present *EvoGraph*, a highly efficient and scalable GPU-based dynamic graph analytics framework that incrementally processes graphs on-the-fly using fixed-sized batches of updates. To realize this vision, we propose a programming model called I-GAS that is based on the gather-apply-scatter programming paradigm and that

allows for implementing a large set of incremental graph processing algorithms seamlessly across multiple GPU cores. Further we propose novel optimizations like property-based dual path execution in the EvoGraph framework to choose between an incremental vs static run over a particular update batch and GPU 'context merging' to efficiently use of all hardware resources and avoid context switching overhead using GPU streams. Extensive experimental evaluations for a wide variety of graph inputs and algorithms demonstrate that EvoGraph achieves substantial speedup compared to static graph recomputation and other competing streaming graph processing frameworks such as STINGER.

CHAPTER I

INTRODUCTION

High performance machines are increasingly using GPUs [31, 78, 90, 110, 111], to leverage their scalability and low dollar to FLOPS ratios. As a result, GPUs have become the main compute engines for todays HPC clusters and supercomputers like the Titan supercomputer in Oak Ridge [18]. This trend continues with the move toward exascale machines [101, 41, 36, 54, 35, 73, 38], with compute nodes expected to be comprised of millions of accelerator and general purpose cores, whether packaged as 'thin' or 'fat' nodes (shown in Figure 1). Therefore, it is not only important to efficiently schedule applications to keep all the available cores busy but also intelligently move the appropriate data near the computation as these accelerators have limited amount of memory attached to them. Existing software infrastructures deal poorly in terms of scheduling and fine grain resource management of such heterogeneous architecture leading to substantial underutilization of all the available resources, including both the computational and data movement engines. Next we describe three broad classes of applications where there is substantial GPU underutilization and showcase the current scheduling challenges in them.

1.1 GPU Sharing in Cloud

A recent trend is the gain in popularity of computationally intensive high performance applications in client-server workloads including image processing algorithms like video transcoding [9] , financial algorithms [26], online gaming, e.g., NVIDIAs cloud gaming [12], search [68], data mining [72] and multimedia services like Adobes Photoshop.com [1]. This motivates online services to take advantages of GPU clusters. This is mirrored by GPU offerings by cloud providers like Amazon ECC [2], Nimbix [11], Peer1 Hosting [15], and Penguin Computing [16]. Figure 2 shows the application service model for a multi-tenant single GPU server. User requests follow a negative exponential distribution and are served by a finite number of server threads. The exponential distribution models intermittent

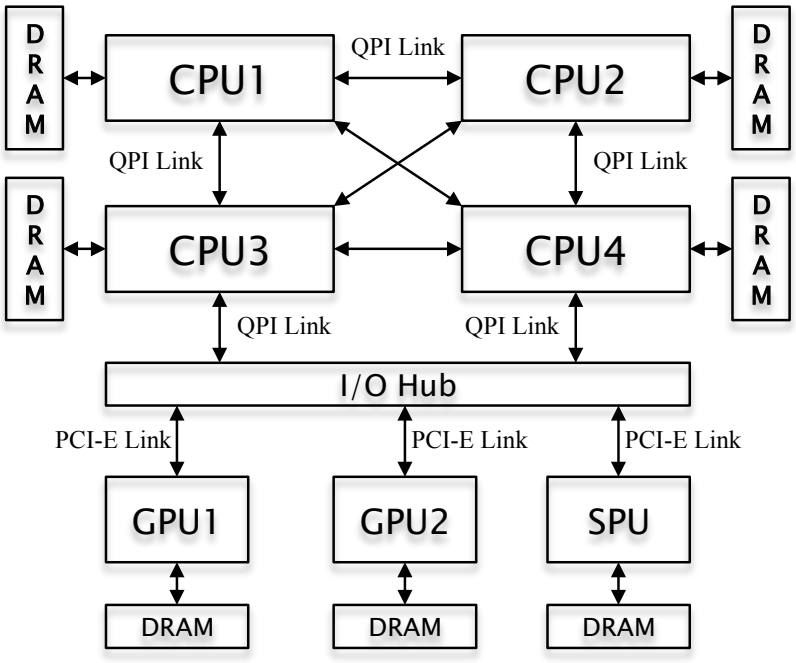

Figure 1: Accelerator-based heterogeneous system architecture

periods of bursts of load when application requests queue up while other requests are being processed, followed by periods of calm when the accumulated requests are serviced.

A challenge to using GPUs in these multi-tenant server and cloud environments is the lack of support for sophisticated GPU scheduling, given the predominant model of treating GPUs as statically scheduled devices, in which applications explicitly and programmatically select the GPU devices on which they wish to run, rather than as first class schedulable entities [60]. Such static GPU assignments will inhibit concurrency, particularly with the varying workloads imposed by web applications. For instance, during peak demands for certain services, some GPU devices will be heavily utilized while for other services GPUs will be idle or underutilized. Low GPU utilization can also be attributed to considerable diversity in the fraction of CPU vs. GPU component in applications, for reasons that include an inability to parallelize certain application components and/or limited GPU residency vs. the costs of host-GPU data transfers. Finally, although each GPU can internally contain thousands of cores, it is treated by applications as a single SIMD engine, potentially resulting in the serial execution of GPU contexts that could have been executed concurrently.

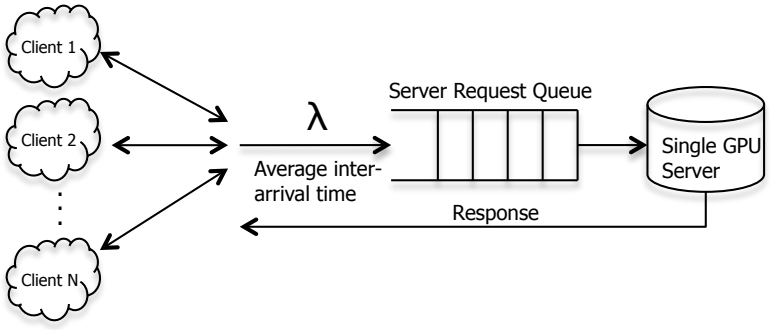

Figure 2: GPGPU application service model following a negative exponential distribution of request arrival from multiple end users.

1.2 Large-Scale Graph Analytics on GPUs

With the increasing interest in many emerging domains such as social networks, the World Wide Web (e-commerce and advertising), and genomics, the importance of graph processing has grown substantially. Some examples of graph analytics include friend/product recommendations [105], anomaly and trend detection [113], online advertisement serving [58] etc. This recent trend has given rise to many graph processing frameworks in both distributed, e.g. GraphLab [77], PowerGraph [56], Pregel [80]; and single machine shared-memory environments, e.g. Graphchi [71], X-Stream [93], Ligra [103] etc. This need to rapidly process large graph-structured data has also engendered recent efforts to leverage cost-efficient GPUs for efficient graph analytics. Doing so, however, requires addressing substantial technical challenges, including (1) dealing with the dynamic nature of graph parallelism, (2) coping with constrained on-GPU memory capacity, i.e., to process graphs with memory footprints that exceed that capacity, and (3) addressing programmability issues for developers with limited insights into how to best exploit the resources of evolving and varied GPU architectures.

More precisely, a graph processing framework using GPUs should expose abstractions or simple APIs for the developers to write the appropriate sequential codes for their domain specific algorithms, e.g., for data mining, machine learning, etc to express their use for processing graphs of arbitrary size. The runtime should then seamlessly (i) partition the graphs into smaller chunks each single one of which entirely fits into GPU memory,

(ii) efficiently move data between host and device leveraging concurrent GPU operations to obtain fine-grain parallelism that exploits both GPU software and hardware features like CUDA streams and Hyper-Q of Kepler GPUs etc (iii) choosing the most appropriate programming model (edge- or vertex- centric or a combination of both) to generate device code for efficient GPU-level graph processing, and iv) finally intelligent coordinated scheduling and management of both the data movement and compute engines to achieve optimal performance. With such automation, we can deal with graph sizes much exceeding GPU memory sizes. This is important because even a common Yahoo web-graph comprised of 1.4 billion vertices requires approximately 6.6 GB of memory to store just its vertex values (not even including the edges and their corresponding states).

In summary, the goal is to design a scale-up graph processing framework on HPC systems with discrete GPUs and high end (i.e., memory-rich) hosts where GPUs can be used to accelerate analytics performed on graphs with billions of edges, operating at speeds much exceeding that of similar operations run on CPUs, and programmed in ways accessible to programmers who are not experts in GPU programming.

1.3 Dynamic Graph Analytics on GPUs

Another important aspect of real-world graphs like Facebook friend lists or Twitter follower graphs is that they dynamically change with time. Current graph analytics on such dynamic graphs follow a store-and-static-compute model that involves first storing batches of updates to a graph applied at different points in time and then repeatedly running static graph computations on multiple versions or snapshots of this evolving graph sequence. The key assumption made here is that the rate of change in graphs due to continuous updates is slower than the execution time of the static graph analytics. This assumption might not hold true for current real-world graphs. For instance, Twitter traffic can peak at 143 thousand tweets (and associated updates) / per second and emails sent per second can reach as high as 2.5 millions/sec. Hence, there are two fundamental challenges to applying static recomputation to these types of rapidly changing data sets. First, static graph analytics on a single version of the evolving graph, even when leveraging massive amount of parallelism

offered by multiple cores in a high performance cluster, can be very slow due to the extreme scale of many real-world graphs and/or because of the complexity of the graph queries that are traditionally both compute and memory intensive. Therefore, the cumulative cost of analyzing such large-scale versions with complex graph queries repeatedly can be substantially high. Second, there are real world graph analytics problems that inherently require soft or hard real time guarantees, e.g., real-time anomaly detection, disease spreading etc. So to conclude, the current high volume and velocity of graph data combined with the complexity of user queries has outstripped the traditional static graph analytics model on streaming graphs.

To address the above mentioned computational challenges in dynamic graph processing we need a graph processing framework that can incrementally process a continuous stream of updates (i.e., edge/vertex insertions and deletions) as a sequence of batches. Because the incremental logic, in many practical scenarios, affects only a portion of the graph, this reduction can result in large performance benefit compared to static recomputation of the graph algorithm on the entire graph for many popular graph algorithms and real-world graphs. Further, we also need to handle the scenarios when updates to the graph affects a very large portion of the graph and incremental processing won't help much or may even be worse (due to overheads of incremental execution) compared to a static recomputation. E.g., in incremental Breadth First Search (BFS), updates that affect vertices close to the root node affect nearly the entire BFS tree. In this case, the incremental run can at best perform as good as the static re-run. Hence a characterization of graph algorithm that would benefit the most from incremental processing is essential. Finally, in order to allow faster updates to the graph and run both the incremental and static graph algorithms efficiently on GPUs, we need to design appropriate data structures specifically tailored to store both the graph and the updates for efficient scheduling and data movement between the host and the device.

1.4 Thesis Statement

The future exascale machines with compute nodes are expected to be comprised of millions of heterogeneous accelerator-based and general purpose cores. It is a huge challenge to efficiently schedule applications and place the appropriate data near the computation. To address this resource management and scheduling challenges we must build system-level design and abstractions that support load balanced scheduling of application requests to avoid request collisions, feedback-based mechanisms for efficient data movement and placement, system-level support for reducing core idling and seamlessly scaling to large input datasets, particularly those arising from the processing of complex GPU-based applications like graph analytics.

1.5 Contributions

The key contribution of our research is a set of technologies that addresses the aforementioned challenges. Specifically, to validate the thesis, we make the following contributions:

- **Strings Scheduler.** To address the challenges in scheduling multi-tenant cloud workloads in the heterogeneous resources of future, high-end manycore GPU-based server platforms we design and implement the Strings scheduler, a two-level hierarchical scheduler that decomposes the scheduling problem into a combination of load balancing and per-device resource sharing. The workload balancing intelligently binds each applications GPU component to an appropriate GPU and the device-level, per-GPU scheduler handles GPU resource sharing for the multiple tenants mapped to a single GPU, to improve application performance while also meeting system-level goals like high throughput, fairness, etc. It implements a model in which accelerators like GPUs are first class schedulable entities rather than statically chosen devices used as single SIMD engines. The intent is to avoid the serial execution of GPU contexts that could otherwise have been executed concurrently, as with the varying workloads imposed by web applications, where during peak demands, the current model of static GPU assignments will cause some GPU devices to be heavily utilized while others are idle or underutilized. Explicit scheduling can also avoid underutilization caused

by the differences in the fraction of CPU vs. GPU components seen across different applications, for reasons that include an inability to parallelize certain application components and/or limited GPU residency vs. the costs of host-GPU data transfers. Strings makes GPUs into explicitly scheduled entities by overriding the device selection calls made by applications. It then manages these calls with the aforementioned two-level scheduler, at the top, balancing workloads across the multiple GPUs resident in each manycore node, and at the device level, reducing GPU core idling via GPU multi-tenancy and the judicious overlap of GPU execution with host-GPU data movements. Strings also supports true GPU multi-tenancy, termed the 'Context Packing', which dynamically packs the GPU contexts of multiple applications into a single context, to achieve high GPU utilization and low context switching overhead. Additional methods enable in providing dynamic feedback from device-level schedulers to workload balancer, to inform the global decisions made by the latter about characteristics of the applications being scheduled by the former.

- **GraphReduce Framework.** To address the problem of processing graph applications with larger memory footprint than the device memory, we present GraphReduce (GR), a highly efficient and scalable GPU-based out-of-core graph processing framework that operates on graphs that exceed the devices internal memory capacity. GraphReduce supports an access pattern based hybrid computational model adopting a combination of edge- and vertex-centric implementations of the Gather-Apply-Scatter (GAS) programming model to match the different types of parallelism present in different phases of the GAS execution model. GR achieves efficiency in graph processing via improved asynchrony in computation and communication (operating on multiple asynchronous GPU streams), by dynamic characterization of data buffers based on data access pattern and access locality to fully exploit the high degrees of parallelism in GPUs. Additional hardware parallelism is extracted via *spray streams* for deep copy operations on separate CUDA streams. GR runtime also uses computational frontier information for efficient GPU hardware thread scheduling and data movement between host and GPU. Specifically, GR moves data into GPU memory

only when a subset of the graph has at least one active vertex or edge. Further, when possible, GR uses dynamic phase fusion/elimination to merge/eliminate multiple GAS phases, to avoid unnecessary kernel launches and associated data movement.

- **EvoGraph.** Because of the extreme scale of real-world graphs and the high rate at which they evolve combined with the complexity of user queries, traditional processing methods by first storing the updates and then repeatedly running static graph analytics on a sequence of snapshots are deemed undesirable and computational infeasible on GPUs. To address such challenges of processing real-world graphs that are constantly evolving over time we present the design and implementation of EvoGraph, a high performance dynamic graph analytics framework for evolving graph analytics on GPUs that incrementally processes graphs on-the-fly using fixed-sized batches of updates. As part of EvoGraph, we propose a novel programming model called I-GAS that is based on the gather-apply-scatter programming paradigm and that allows for implementing a large set of incremental graph processing algorithms seamlessly across multiple GPU cores. We further propose novel optimizations like property-based dual path execution in the EvoGraph framework to choose between an incremental vs static run over a particular update batch and GPU 'context merging' to merge and collocate the GPU contexts of static and incremental graph algorithms on the same GPU, inorder to avoid context switching overhead and efficiently use of all hardware resources using GPU streams, including its computational and data movement engines.

- Extensive performance evaluation of each of the above runtime frameworks on wide variety of workloads and algorithms to demonstrate their effectiveness when compared to state-of-the-art competing solutions.

1.6 Dissertation Structure

The rest of this dissertation is organized as follows.

Chapter 2 discusses the design and implementation of Strings, a hierarchical scheduling

framework for efficient sharing and scheduling of multi-tenant cloud workloads on multi-GPU server systems.

Chapter 3 presents the design and implementation of Graphreduce, a framework for large-scale out-of-core graph processing using GPUs where the input graph may or may not fit in GPU memory, supporting access pattern based hybrid computational model and efficient data movement techniques.

Chapter 4 explains in detail the design and implementation of EvoGraph, a high performance dynamic graph processing framework for evolving graph analytics using GPUs that incrementally processes graphs on-the-fly using fixed-sized batches of updates.

Each of the above chapters also include a detailed performance evaluation of each of the runtime frameworks presented above on wide variety of workloads and algorithms validating their effectiveness when compared to state-of-the-art competing solutions.

Chapter 5 discusses the salient research related to the systems and topics dealing with graph processing, including accelerator-based and real-time streaming graph processing.

Chapter 6 concludes the dissertation and presents future avenues of research.

CHAPTER II

STRINGS: MULTI-TENANCY IN ACCELERATOR-BASED SERVERS

Cloud and server infrastructures routinely use GPUs to service computationally intensive client workloads, for online gaming [12], multimedia services [9] and image processing [1], financial codes [26], data mining [72] and search [68], and to support the needs of next generation applications like perceptual computing [27, 39, 49, 24, 86]. This trend is mirrored by GPU offerings by cloud providers like Amazon ECC [2], Nimbix [11], Peer1 Hosting [15], and Penguin Computing [16].

The effective use of GPUs in these multi-tenant server and cloud infrastructures, however, challenges the current model of static GPU provisioning, in which applications explicitly and programmatically select the GPU devices on which they wish to run. Such static GPU assignments will inhibit concurrency, particularly with the varying workloads imposed by web applications. For instance, during peak demands for certain services, their GPU devices will be heavily utilized while other services' GPUs will be idle or underutilized. Additional GPU underutilization will be caused by application-specific variations in their fraction of CPU vs. GPU execution time, for reasons that include an inability to parallelize certain application components and/or limited GPU residency vs. the costs of host-GPU data transfers.

The *Strings* scheduler described in this chapter adopts a multi-tenant model in which accelerators like GPUs are treated as first class schedulable entities [66, 59, 89, 60], by overriding the device selection calls made by applications and then managing their GPU calls with a two-level scheduler: at the higher level, on each platform, balancing workloads across the multiple GPUs attached, and at the GPU device level, reducing core idling via multi-tenancy and the judicious overlap of GPU execution with host-GPU data movements. Additional performance improvements are derived from dynamically merging the GPU contexts of different applications and by providing dynamic feedback about fine-grain

application characteristics from device-level schedulers to the workload balancer.

Using *Strings* with workloads drawn from diverse classes of cloud applications, this chapter presents and evaluates GPU scheduling policies distinct from prior work in their explicit consideration of data movement to/from the GPU device. (1) The Phase Selection (PS) policy co-schedules on the same GPU those applications that currently operate in different phases computation vs. communication – of their combined CPU/GPU execution. Using PS results in an average speedup of **6.41x** over static provisioning with the CUDA runtime. (2) Advanced feedback-based policies, termed Data Transfer Feedback (DTF) and Memory Bandwidth Feedback (MBF), capitalize on the advantages offered by CUDA streams and by Strings built-in support for merging GPU contexts belonging to different applications. DTF collocates applications with contrasting data transfer times to maximize the concurrent use of a GPU's memcpy vs. compute engines. MBF improves overall performance by concurrently executing and hiding the large memory latencies seen by a memory bound application by switching to a compute bound application. DTF and MBF achieve notable improvements in average system throughput, by **8.06x** and **8.70x**, respectively, compared to the commonly used CUDA runtime. (3) Further improvements in performance are derived from dynamic changes to the workload balancing policies being used in response to device-level observations of altered behavior in the GPU tasks being run.

This chapter makes following technical contributions:

- A two-level hierarchical scheduler where workload balancing intelligently binds each applications GPU component to an appropriate GPU, along with a device- level, per-GPU scheduler that handles GPU resource sharing for the multiple tenants mapped to a single GPU, to improve application performance while also meeting system-level goals like high throughput, fairness, etc.

- Support for multi-tenancy, termed the Context Packer, which dynamically packs the GPU contexts of multiple applications into a single context, to achieve high GPU utilization and low context switching overhead.

- Dynamic feedback from device-level schedulers to workload balancer, to inform the

global decisions made by the latter about the characteristics of the applications being scheduled by the former.

- A novel GPU scheduling policy, called Phase selection (PS), which maximizes the concurrent use of a GPUs memcpy vs. compute engines, by smartly selecting applications currently running in different phases of their combined CPU/GPU execution.

- Advanced feedback-based policies like DTF and MBF that exploits the advantages offered by CUDA streams by collocating applications with contrasting behavior, in terms of data transfer and memory intensity, to achieve extreme performance benefits.

2.1 Background and Motivation

2.1.1 Scheduling Challenges in GPU Multitenancy

Current programming models continue to treat GPUs as devices chosen by applications. There are several issues with the consequent programmer-defined selection of target GPUs. First, applications running on a multi-GPU node may compete for the same GPU, thus not able to leverage availability of multiple on-node GPU accelerators and leading to the serialization of GPU requests that otherwise could have been served in parallel. We define such conflicts as *static collisions* between applications' GPU requests. Second, since applications are unaware of each others GPU usage, e.g., their relative GPU intensities, they cannot assess the performance implications of sharing a single GPU. We define this as a *character collision* between the requests from two or more applications sharing a GPU. Both static and character collisions become even more critical when nodes have heterogeneous GPUs with differing capabilities in terms of their compute, memory capacities, and bandwidths.

The importance of collisions is underlined by the fact that most cloud applications driven by end user requests vary substantially in their compute and memory characteristics and therefore, have difficulties in fully utilizing both the compute engines and memory capacities of GPUs. We demonstrate this in Figure 3, with cloud applications deployed using the CloudBench [104] infrastructure, for exponentially distributed request arrivals. The color-coding in the figure indicates the levels of compute and memory utilization of the applications, varying from heavily utilized (red $> 90\%$) to under-utilized (green $< 10\%$).

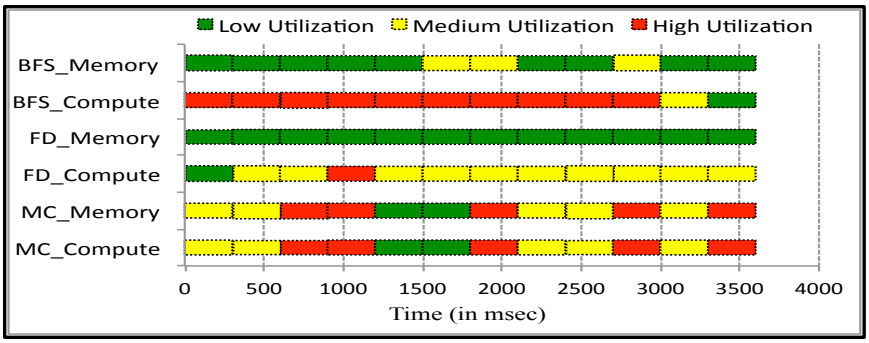

Figure 3: Compute and memory characteristic of various GPU-based cloud applications.

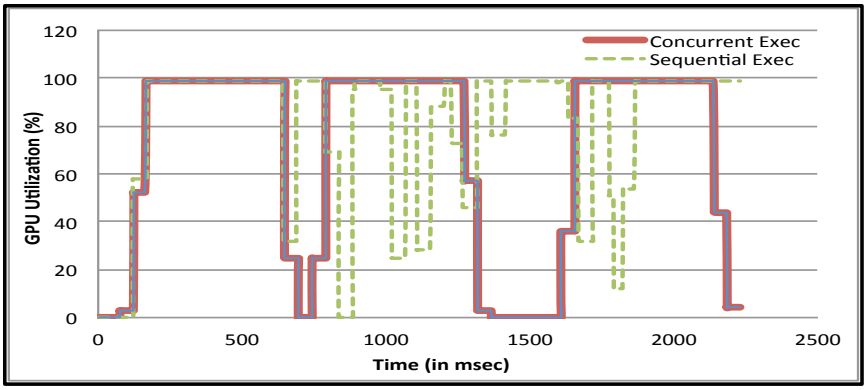

Figure 4: GPU utilization of Monte Carlo requests following exponential distribution of request arrival with sequential vs. concurrent execution.

Some of these applications are compute intensive, such as graph algorithm Breadth First Search (BFS), some are memory intensive financial algorithm Monte Carlo (MC), and some exhibit average utilization levels, like OPENCV [14] face detection (FD). Note that frequent GPU idle intervals occur even for efficient GPU codes like Monte Carlo.

Another issue with current GPU programming models is that although each GPU can internally contain thousands of cores, application uses it as a single SIMD engine [115], which means that the multiple GPU contexts created by host threads can share a GPU only over time, but not in space. CUDA 4.0 addresses this problem by allowing multiple threads within a single host process to share the same GPU context, but GPU utilization could be improved further with true GPU multi-tenancy. We demonstrate this opportunity by manually dispatching multiple sets of independent Monte Carlo requests, again following an exponential distribution of inter-arrival times, over different CUDA *Streams* [4] from the same GPU context. Figure 4 shows GPU usage peak and idle periods to be much

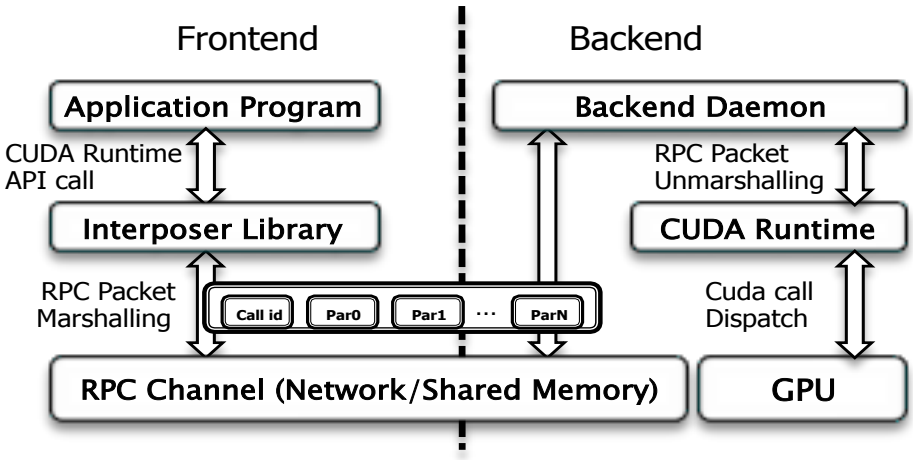

Figure 5: Architecture of GPU Remoting.

more uniform compared to their sequential execution. This is because keeping a single GPU context avoids context switching overhead and this eliminates unnecessary GPU idling during context switching (the 'glitches' in the figure), as in the case of independent sets of web requests driving their execution.

The illustrative examples above motivate key properties of the Strings approach to effective multi-tenancy in GPU-based servers: (1) load balancing is needed to avoid static collisions, (2) device-level scheduling must be cognizant of character collisions and provide such feedback to the load balancer, (3) additional functionality is needed to achieve resource management goals like fairness, high throughput, etc., and (4) there should be system-level support for reducing GPU core idling when some application's context cannot fully utilize a single GPU. We next describe the Strings infrastructure and its utility for realizing and experimenting with effective scheduling strategies for cloud and multi-tenant workloads using GPUs.

2.2 System Design Principles

2.2.1 Future GPU Servers and gPool

Scheduling the potentially multiple GPUs in future server platforms demands (i) the logical aggregation of all GPUs to make them visible to the scheduler and then (ii) decoupling the CPU-GPU associations programmed into GPU-based applications. Strings adopts from

previous work (e.g., GVim [59], vCuda [102], rCuda [44], Pegasus [60], gVirtus [55]) an API-driven separation of an application's CPU from its GPU components. As shown in Figure 5, (i) a *frontend* implemented as a CUDA runtime interposer library dynamically links with the application, responsible for intercepting the CUDA runtime API calls, and (ii) a *backend* is realized as a daemon responsible for receiving GPU requests from the frontend, dispatching the CUDA runtime library calls to the attached GPUs, and returning error codes and/or output parameters to the frontend. A useful side effect of this architecture is the ability to execute an application's GPU component on a GPU attached to some remote node, termed GPU remoting [83]. We do not explore this topic at scale, but use it to create a supernode, which is an emulated high-end multi-GPU server machine with more GPUs than those available on today's single physical platforms. For such a supernode, Strings aggregates all GPUs into a single logical pool, termed a *gPool*, for use by the GPU scheduler. The gPool is formed when the GPU virtualization runtime is started and the backend daemons are spawned in each participating node. Each backend collects the information of GPUs in its own node and sends it to the *GPU Affinity Mapper*, discussed later. It assigns a unique GPU id (GID) to each of the GPUs in the pool, builds a mapping from GID to `<node_id (IP address), local device_id>` pair, called the *gMap*, and broadcasts it to all participating machines.

With gPools, gMap, and GPU request interposition, the *Strings* scheduling infrastructure described in this chapter permits any node in the gMap to participate in GPU scheduling. Figure 6 shows the logical transformation of a small number of machines with per-node GPUs into a single supernode with sets of GPUs schedulable via a shared GPU pool. The experimental results in this chapter are obtained with a dual-machine supernode connected via dedicated network links. This purposely small scale setup makes it possible to treat remote GPUs much like NUMA memory is treated in high end servers, ignoring issues like network contention likely to occur for scaleout systems [83].

2.2.2 Design Decisions

1) Efficient and Robust Scheduling

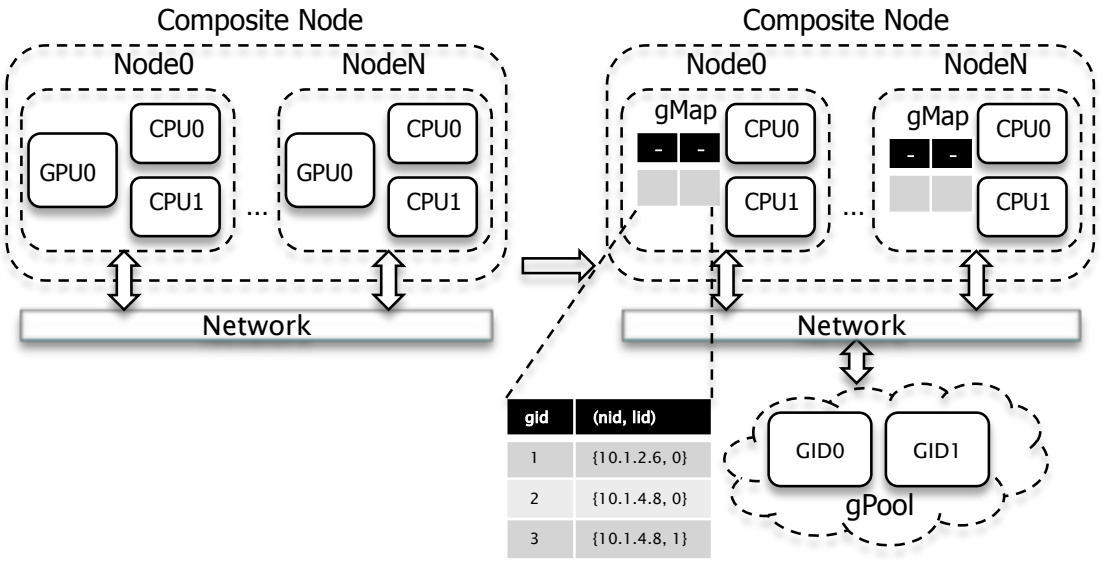

Figure 6: Logical transformatiom of GPU cluster after gPool creation.

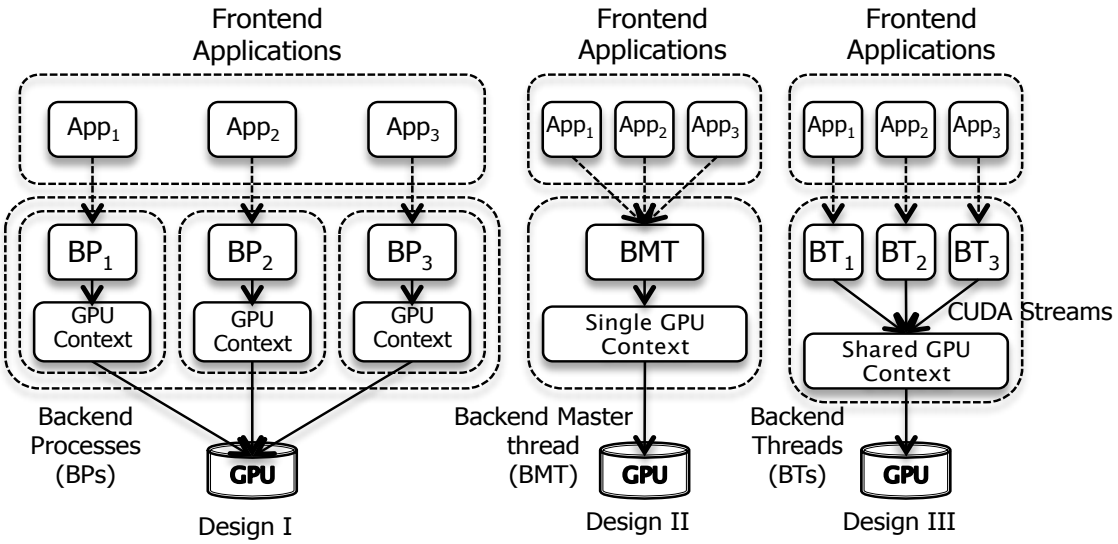

Figure 7: Three different implementations GPU remoting.

The frontend/backend model suggests three methods for mapping frontend applications to backend workers (processes or threads), shown in Figure 7.

Design I. Each frontend application is mapped to a unique backend process, which then dispatches the actual accelerator (e.g., CUDA) calls to some physical GPUs. This design offers high fault tolerance and security to frontend applications, as it isolates their GPU components in separate backend protection domains (GPU contexts). While used in

our previous 'Rain' scheduler [96], as indicated in the figure, a drawback is that a large number of frontend applications will require an equally large number of backend processes, hurting scalability. For NVIDIA GPGPUs, because the CUDA runtime does not allow two different host processes to share the same GPU context, the GPU components of two different applications cannot run concurrently on a single GPU, resulting in GPU context switching overhead and potential GPU core idling.

Design II. An alternative design avoids context switching, by packing different application contexts into a single protection domain [76], which we term 'context packing'. Specifically, by mapping each frontend application to a different CUDA Stream [57, 89], the design creates a single backend thread per device, thus consolidating the GPU components of all frontend applications into a single hosted GPU context. Advantages include (i) minimal backend context switching overheads, reduced further by pinning the per GPU backend threads to certain CPU cores, and (ii) the presence of a single GPU context hosting all frontend applications, which enables the cross-application space-shared use of GPU resources multi-tenancy. Such efficient GPU space sharing is useful for multi-tenant cloud workloads less concerned with isolation (e.g., Amazon's web store runs multiple web servers in a single VM, for efficiency in resource usage). It is also useful for pairing applications with different characteristics, e.g., one with high memory bandwidth, the other highly compute intensive, but with their aggregate GPU resource requirements not exceeding those available in the physical GPU. An advantage specific to CUDA is (iii) that by leveraging CUDA streams, all three GPU engines, (a) memory copy from host to device (H2D), (b) from device to host (D2H), and (b) compute, can be concurrently used by different applications, to fully utilize these GPU resources. Potential shortcomings of the design are that (1) it is susceptible to faults, e.g., if the master thread managing all requests to a particular GPU crashes, all frontend applications relying on it are affected, (2) a malicious application can corrupt the entire GPU context or gain unauthorized access to other applications data, (3) the single master thread has to continuously synchronize with all frontend applications to ensure fair overall progress and pipelined execution, which significantly adds to the complexity and overhead of the runtime, and (4) a blocking call, e.g., cudaDeviceSynchronize(), made

by one application will block all other applications sharing the same GPU context, and deferring such call for a long time will lead to application starvation.

Design III. Strings adopts a hybrid of Designs I and II, leveraging the fact that for NVIDIA GPUs, from CUDA v4.0 onwards, GPU contexts are hosted per process per device, which implies that the GPU operations invoked from threads within a single host process can run concurrently on a GPU, while those from separate processes are still multiplexed by the device driver. As shown in Figure 7, in Strings, therefore, the GPU components of all frontend applications sharing a particular GPU are mapped to separate backend threads of the same per GPU backend process, with their respective GPU operations invoked via separate CUDA streams. The design has reduced overhead compared to Design I, due to reduced thread vs. process context switch overheads. While not providing complete isolation, the design improves on Design II in that faults can be localized to certain threads. Most importantly, the GPU operations from different applications can run concurrently, thereby inheriting all of the benefits of space and time sharing of Design II. Further, as GPU requests are channelized through separate backend threads, overheads of request synchronization and of pipelined execution are reduced to a minimum, and properties like fair progress for GPU applications are much easier to implement.

2) Asynchronous Operation

The presence of an explicit interposer affords additional optimizations. First is the removal of blocking calls, by converting all device synchronization calls to their respective stream synchronization counterparts, e.g., cudaDeviceSynchronize() converted to cudaStreamSynchronize(). This ensures that all the applications sharing a GPU, under the umbrella of a single GPU context, are not stalled when one application explicitly synchronizes with the device.

Second is the runtime conversion of all synchronous memcpy operations into their respective asynchronous versions. With this optimization, (i) subsequent CUDA calls that are not dependent on the memcpy operation can proceed without waiting for memcpy calls to finish, (ii) we hide the overhead introduced by the runtime due to interposition, marshaling, RPC, unmarshalling etc, by allowing execution to proceed and overlap even for

calls that are dependent on the memcpy operation. For instance, a cudaLaunch() call that depends on a memcpy typically has to wait for the memcpy to complete, but because it is now asynchronous, the runtime layer overhead for cudaLaunch() can be overlapped with the asynchronous data transfer to the device.

Finally, asynchrony can also be achieved for hidden and synchronous runtime API calls that do not have output parameters, by making interposer-based RPCs non-blocking. This does not violate the correctness in single threaded applications as RPC requests from within an application remain in-order but might affect multi-threaded applications where RPCs from separate threads are not guaranteed to be in-order e.g., when cudaLaunch() from one host thread depends on a memcpy from another and an asynchronous RPC makes the former call to be dispatched before the latter. The problem can be corrected with per-device buffer synchronization logic that maintains the application-intended order of GPU operations across multiple threads within a single application.

2.3 Strings Architecture

We next describe the two-level Strings scheduling infrastructure that avoids static and character collisions of GPU requests, efficiently utilizes the underlying GPU cores with minimum GPU context switching overhead, and meets system goals like throughput and fairness (see Figure 8).

GPU Affinity Mapper - Workload Balancing. To minimize static collisions, the Strings runtime overrides the application's target GPU selection calls, replacing them with decisions made by the GPU affinity mapper/workload balancer. The lifetime of the Strings target device selection call is as follows: (i) an application's cudaSetDevice() call is intercepted by the interposer and forwarded to the workload balancer; (ii) its GPU selection based on static (device capabilities) and dynamic (GPU load, application type, feedback from lower scheduling layer) parameters is returned as a global GPU id (GID) to the interposer; (iii) the interposer uses the GID to acquire a node id and local GPU id from the gMap, and (iv) using GPU remoting, it then forwards the call to some appropriate backend process to bind with the target GPU; (v) the binding is removed when the application exits

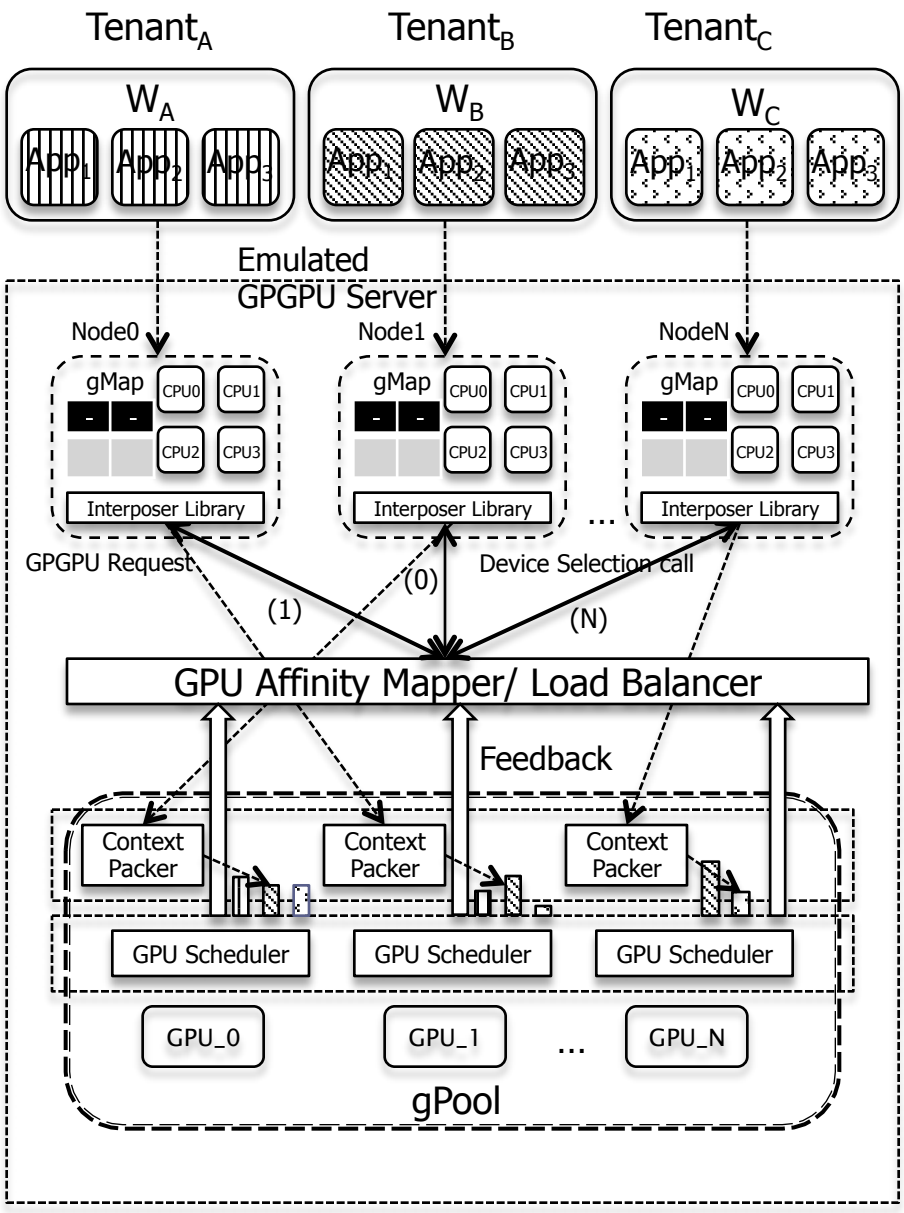

Figure 8: Software architecture of Strings.

or calls cudaThreadExit(). The *GPU Affinity Mapper* is also responsible for the cluster-wide aggregation of GPUs through gPool creation. Shown in Figure 9, it has the following components:

- ***gPool Creator (GC):*** during system initialization, the GC collects device information from the backend daemons of each node in the cluster, assigns a GID to every GPU, creates the gMap and broadcasts it to every node. GC is also responsible for

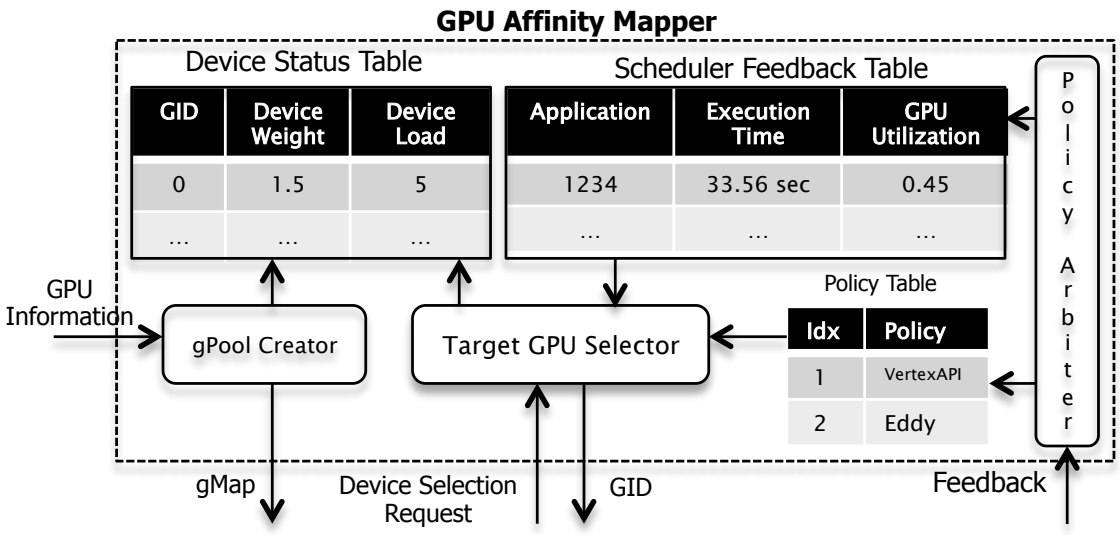

Figure 9: The structure of the GPU Affinity Mapper.

the one time assignment of relative weights to all GPUs based on the device property information received and updating a global data structure, Device Status Table (DST), with this static information. DST also maintains the dynamic states (e.g. current device load) of all the GPUs in the system, which is updated by TGS (explained later) as GPU requests arrive.

- ***Policy Arbiter (PA):*** using the feedback mechanism, the PA receives information about application characteristics like execution time, GPU utilization, data transfer time etc. from the Feedback Engine (FE) of the device-level GPU schedulers and updates a history-based table, Scheduler Feedback Table (SFT), that stores such fine-grain device specific application characteristic information. The PA also triggers dynamic policy switching, upon receiving sufficient feedback information from low-level GPU schedulers.

- ***Target GPU Selector (TGS):*** as the core of the workload balancer, it selects an appropriate GPU for an application. Based on the information in the DST and SFT, for each GPU selection request, it computes the target GID using the selected scheduling policy from the Policy Table (PT) and then returns it to interposer, which then maps the application to a particular GPU in gPool. The PT contains two classes

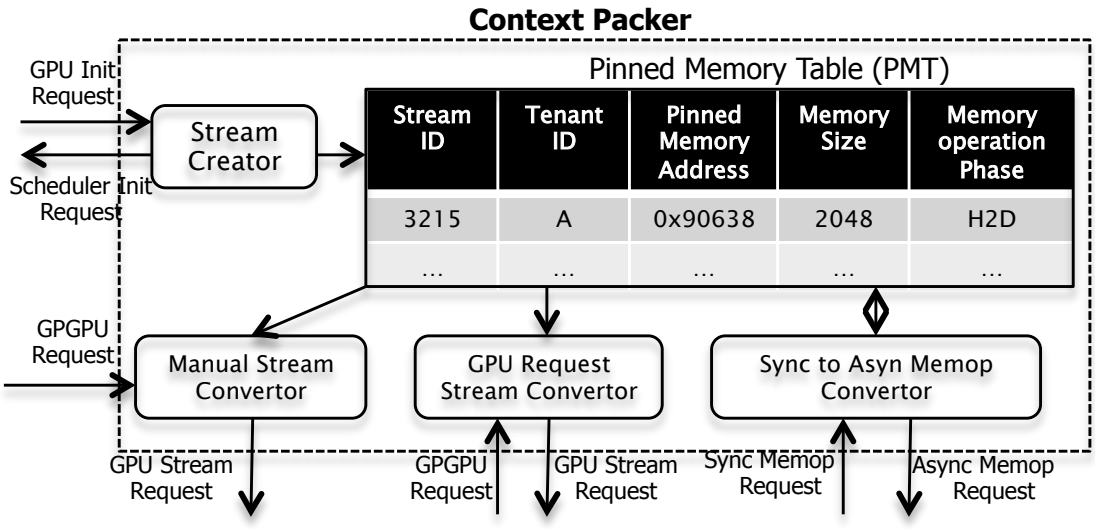

Figure 10: The structure of the Context Packer.

of policies, one that uses only the DST, e.g., GRR, GMin, etc., and the other that uses both the DST and low-level feedback information from SFT, e.g., GUF, DTF, etc.

Context Packer. Operating after workload load balancing has assigned a GPU and before device-level GPU scheduling, the Context Packer (see Figure 10), is responsible for packing multiple applications' GPU components that share a GPU, on the fly, into a single GPU context. It also manages the host side locked memory, the dynamic translation of synchronous memory copies to their asynchronous versions, and the dynamic translation of device synchronization calls to their stream counterparts.

- ***Stream Creator (SC):*** when the first GPU request from an application arrives, SC creates a separate CUDA stream object for it, calling cudaStreamCreate(), the handler to which is stored in a thread local storage. Using this handler, subsequent requests from the application are dispatched over the stream. On cudaThreadExit() or application exit, SC tears down the stream by calling cudaStreamDestroy() on the stream handler.

- ***Auto Stream Translator (AST):*** dynamically translates all GPU operations from an application targeted over the default stream (stream 0) to use the stream created

by the SC. E.g., cudaConfigureCall(), when called without an explicit stream handler in its call parameters, is targeted onto stream 0, and AST uses the stream handler for this particular application to translate the call to use that stream.

- ***Sync Stream Translator (SST):*** the GPU calls that synchronize the application and the device are converted to their CUDA stream counterparts by the SST, e.g., cudaDeviceSynchronize() is converted to cudaStreamSynchronize(). This ensures that all of the applications packed into a GPU context associated with a particular GPU are not blocked when one of them explicitly tries to synchronize its host thread with the device.

- ***Memory Operation Translator (MOT):*** translates all memory copies to their asynchronous versions using a per device data structure called Pinned Memory Table (PMT). PMT stores the active host and device pointers associated with all such memory copy calls and is also responsible for keeping track of the current memory copy phase (e.g., H2D) of an application, storing the stream handler, the application id, tenant id, etc. MOT allocates host locked memory of the size of the host buffer for every such memory copy operation, copies the content of the host buffer into it, and stores both the host and device pointers in the PMT. It then creates an asynchronous version of the memory copy call (e.g. cudaMemcpyAsync()) using the host pointer. On the applications next device synchronization call or a device to host memory copy, the MOT searches for the device pointer in the PMT and frees the corresponding host memory. When the application invokes the cudaThreadExit() or exits, the MOT frees all outstanding active host pointers in the PMT associated with the application.

GPU Scheduler. As shown in Figure 11, this per device software layer addresses inter-application interference arising due to the co-location of multiple applications' GPU components on a single GPU. It prioritizes and dispatches GPU requests to physical GPUs in order to meet resource management goals like system throughput, fairness, etc. It is also responsible for monitoring applications bound to the device and sending feedback about their characteristics to the workload balancer.

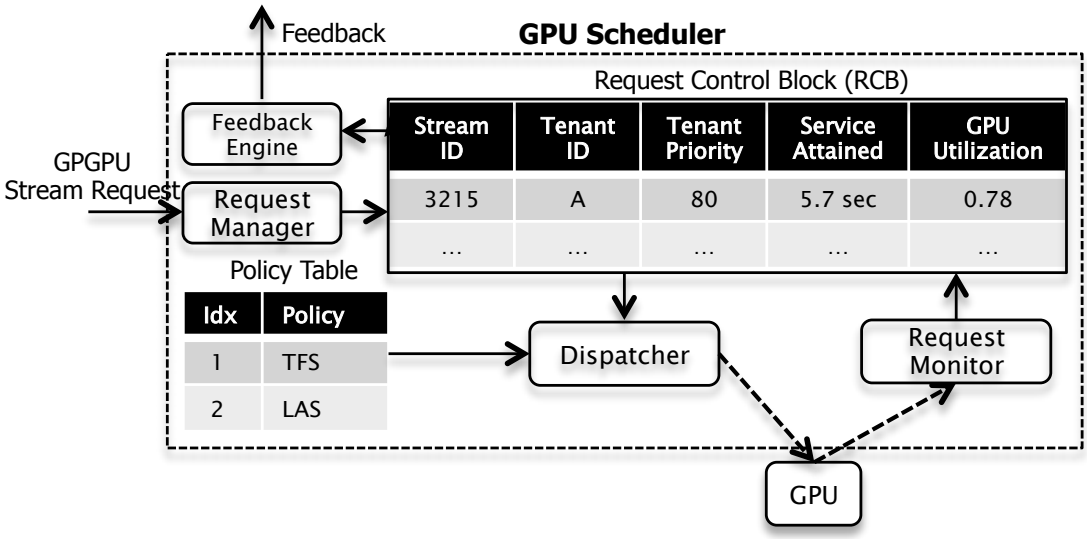

Figure 11: The structure of the GPU Scheduler.

- ***Request Manager (RM):*** registers and unregisters application requests with the GPU scheduler. After the GPU affinity mapper selects the target GPU for an application, the interposer library makes a cudaSetDevice() call to the selected GPU using the GPU remoting infrastructure. On receiving the request, the RM registers the application by creating an entry in a per device data structure called Request Control Block (RCB) with stream id, tenant id and application priority, to be used by the GPU scheduler. RCB also maintains application runtime characteristic information, dynamically computed by the Request Monitor discussed later. When the interposer forwards a cudaThreadExit() call, the RM unregisters the application by removing its corresponding entry from the RCB.

- ***Dispatcher:*** prioritizes and dispatches GPGPU requests to the device. It uses the application characteristic information from the RCB and makes scheduling decisions based on the selected policy from the Policy Table (PT). In Strings three policies are implemented: *True Fair Share* (TFS) to achieve system fairness, *Least Attained Service* (LAS) for high system throughput and *Phase Selection* (PS) for a combination of fairness and system throughput.

- ***Request Monitor (RMO):*** computes GPGPU application characteristics and GPU

```
1 function  GRR():
2   mutex_lock()
3     device_tmp = DEV_RR
4     DEV_RR = (DEV_RR+1)%TOTAL_DEVICES
5   mutex_unlock()
6   return device_tmp
7
8 function GMin():
9   min = 0;
10  mutex_lock()
11    for(i=0; i<TOTAL_DEVICES; i++)
12      if(load_per_device[i] < load_per_device[min])
13        min = i
14        load_per_device[min]++
15  mutex_unlock()
16  return(min)
17
18 function GWtMin():
19 // Static GPU weights are assigned
20 // via offline user training
21  min = 0
22  mutex_lock()
23    for(i=0; i<TOTAL_DEVICES; i++)
24      if(load_per_device[i]*gpu_weight[i]
25          < load_per_device[min]*gpu_weight[min])
26        min = i;
27  load_per_device[min]++
28  mutex_unlock()
29  return min
```

Figure 12: Workload Balancing Policies.

resource usage. Both the workload balancer and GPU scheduler make use of this monitoring information in their scheduling decisions. We currently monitor the total execution time, total GPU time, data transfer time, memory bandwidth, application phase and kernel configuration information of an application. The RMO updates RCB in some regular time interval.

- ***Feedback Engine (FE):*** communicates the application characteristic and local GPU state information, collected by the RMO, to the GPU Affinity Mapper. It retrieves this information from the RCB and feeds it to the SFT when the application request completes. Therefore, when a cudaThreadExit() call arrives, FE piggybacks the feedback information along with the return value of the CUDA call and sends it back to the interposer, which then forwards the same to the GPU Affinity Mapper.

2.4 Scheduling Policies

Strings implements a rich set of scheduling policies, to achieve two cloud- and server-centric goals: fairness for multiple tenants, coupled with high overall system throughput.

2.4.1 Workload Balancing Policies

Three workload balancing policies across multiple accelerators are suitable for server systems, driven by external workloads like those seen for cloud and web applications. Figure 12 shows the pseudo code for these workload balancing policies.

- ***Global Round Robin (GRR):*** assigns incoming applications to the GPUs in the gPool in a round robin fashion.
- ***GMin:*** taking into account the differences in application runtimes, GMin enhances GRR by maintaining a record of the number of applications currently bound to a particular device in the *device load* field. GMin chooses the GPU with minimum device load. Because remote GPUs are more expensive to access, GMin breaks ties by giving preference to local GPUs over remote ones.
- ***Weighted-GMin:*** considering heterogeneity across GPUs, in terms of compute, memory capacity and bandwidth, the weighted-GMin (GWtMin) policy extends GMin by assigning relative weights to different GPUs and computing weighted minimum load to select a target GPU.

2.4.2 GPU Scheduling Policies

Once workload balancing has been done, GPU scheduling policies concern fairness- and system throughput.

- ***Least Attained Service (LAS):*** a GPU-based application, after offloading work to the GPU, uses the CPU until it issues a call, e.g., cudaDeviceSynchronise(), that requires it to wait for the completion of previously issued GPU work. The objective of LAS is to minimize this CPU 'stall time' and thereby maximize system throughput, by prioritizing jobs with least attained service time [91]. The policy works by increasing

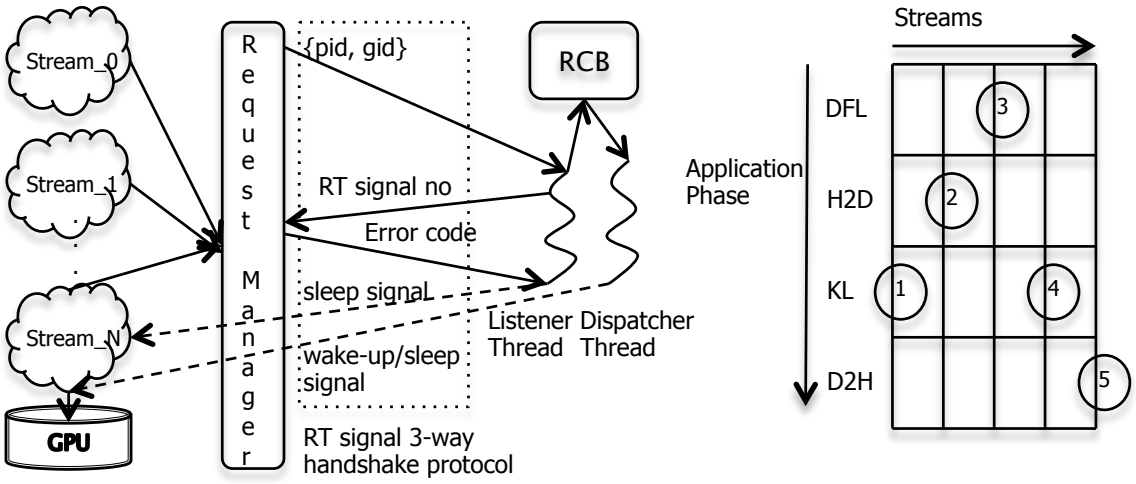

Figure 13: (a) Real-Time Signal based GPU Scheduler (b) Phase Selection Scheduling Policy.

the priority levels of those applications that have attained less GPU service time in a given time quantum. This enables the applications with shorter GPU episodes to finish sooner, thereby reducing overall CPU stall time and improving system throughput. The time quantum chosen in LAS is larger than the time slice assigned to each backend thread by the GPU scheduler, to ensure that an application's GPU characteristic is determined for its long-term behavior. LAS also uses a time decaying GPU service time formula [69], to give higher weight to more recent service epochs.

$$CGS_n = k * GS_n + (1 - k) * CGS_{n-1} \tag{1}$$

CGS_n: *Cumulative GPU service time attained till* n^{th} *epoch*

GS_n : *GPU service time attained in the nth epoch and* $k = 0.8$.

The next set of GPU scheduling policies are implemented using Unix Real-time (RT) signals. Figure 13a shows the three-way handshake protocol followed during the application registration phase: (1) the backend thread corresponding to the GPU application registers its stream id, tenant id, and tenant weight with the RM using IPC; (2) the listener thread of the RM, on receiving the request, creates an entry in the RCB, sends the next available RT signal id to the backend thread; (3) the backend

thread, on receiving it, installs a signal handler and sends the error code as an acknowledgement to the RM. The signal handler registered ensures that the backend thread toggles between its sleep and wake-up states on receiving its assigned RT signal. The Dispatcher, which is responsible for prioritization of GPU requests, uses this mechanism to control which backend threads should be using the GPU and for how long.

- ***True Fair-Share (TFS):*** to ensure fairness among multiple tenants sharing the same GPU, the TFS scheduler ensures a proportionate GPU resource allocation on a per-tenant basis according to their assigned weights. The Dispatcher realizes this by keeping registered backend threads awake only for a time period that is proportional to their tenant weights. The invariant maintained by the Dispatcher is that at any point of time, at most one backend thread is awake and is using the GPU. To address unfairness in GPU access across applications with long vs. short GPU episodes, TFS maintains a history of GPU usage over the past scheduling epochs, and if any application overshoots its allocated time slice, the dispatcher penalizes it in subsequent epochs. Thus, TFS ensures that tenants receive their weighted fair share under high system load and its work-conserving nature distributes a tenants unused shares among the applications of other tenants according to their respective weights.

- ***Phase selection (PS):*** CUDA streams can leverage the parallelism opportunities offered by multiple hardware queues (data transfer and compute) present in NVIDIA GPUs. Specifically, if the GPU scheduler receives a cudaLaunch(), cudaMemcpy() Host to Device (H2D) and Device to Host (D2H) at around the same time from three different backend threads, all of them can be serviced concurrently, as the calls are sent over different streams belonging to the same GPU context. To take advantage of this, the backend threads keep the GPU scheduler apprised of their current GPU usage phase, the Dispatcher identifies threads that are in different GPU phases, and wakes them up. If it cannot find at least one thread from each of the GPU phases, it wakes up threads in the following priority order: Kernel Launch $(KL) > H2D = D2H >$

Default Phase (DFL). Note that this policy relaxes the TFS invariant of keeping only one backend thread awake in any scheduling epoch. The dispatcher picking threads in different phases of their execution has some similarity with playing a guitar chord (Figure 13b), by pressing a set of strings at specific frets. Our GPU scheduling framework derives its name 'Strings' from this analogy.

2.4.3 Feedback-based Load Balancing

It is important to assess accelerator utilization when scheduling its resources, particularly for applications with dynamic usage profiles. Feedback-based policies use such device-level information to guide load balancing. Figure 14 shows the pseudo code for two of the feedback-based policies: *GPU Utilization Feedback* and *Runtime Feedback*. Other feedback-based policies follow similar implementation.

- ***Runtime Feedback (RTF):*** the GPU Scheduler monitors the execution time of requests scheduled on the GPU and provides such feedback to the workload balancer, which uses this to improve future GPU assignments.

- ***GPU Utilization Feedback (GUF):*** the GPU Scheduler provides feedback to the workload balancer about how efficiently an application is using the GPU, by computing GPU utilization, as the ratio of the total GPU time of an application to its total runtime. Borrowing from NUMA-aware thread placement [34], GUF tries to avoid collocation of applications with high GPU utilization on the same GPU. Decisions are refined over time as the system learns about the GPU characteristics of more applications from the feedback mechanism.

- ***Data Transfer Feedback (DTF):*** DTF capitalizes on CUDA streams to overlap device-level computation with data transfers between host and device. By providing feedback to the workload balancer about the time spent on data transfer, it becomes possible to collocate applications with differing characteristics, some being transfer- and others being compute-intensive.

```
function gpu_utilization_feedback(appid1, appid2): //GUF
  feedback_type = FEEDBACK_TYPE_UTIL
  // gpu utilization as a ratio of the total GPU time
  //of an application to its total runtime.
  min = 0
  mutex_lock(&MU)
  for(i = 0; i < TOTAL_DEVICES; i++)
    if(load_per_device[i] * gpu_weight[i]
       < load_per_device[min]* gpu_weight[min])
      min = i

  //example shows the GPU sharing of a pair of
  //applications
  if(all_data_received)
    if(gpu_utilization[appid1][min]
      >= gpu_utilization[appid2][min])
      ratio = (gpu_utilization[appid1][min]
         / gpu_utilization[appid2][min])
      greater = appid1
    else
      ratio = (gpu_utilization[appid2][min]
         / gpu_utilization[appid1][min])
      greater = appid2

    if(greater == appid1)
      load_per_device[min] += ratio
    else
      load_per_device[min] += 1
  else load_per_device[min]++

  mutex_unlock(&MU)
  return min

function runtime_feedback(appid1, appid2): //RTF
  feedback_type = FEEDBACK_TYPE_EX_TIME;
  min = 0;
  mutex_lock()

  for(i = 0; i < TOTAL_DEVICES; i++)
    if(load_per_device[i]  < load_per_device[min])
      min = i

  //example shows the GPU sharing of a pair of
  //applications
  if(all_data_received)
    if(total_execution_time[appid1][min]
       >= total_execution_time[appid2][min])
      ratio = (total_execution_time[appid1][min]
         / total_execution_time[appid2][min])
      greater = appid1
    else
      ratio = (total_execution_time[appid2][min]
         / total_execution_time[appid1][min])
      greater = appid2

    if(greater == appid1)
      load_per_device[min] += ratio
    else
      load_per_device[min] += 1
  else load_per_device[min]++

  mutex_unlock()
  return min
```

Figure 14: Two Feedback-based Policies.

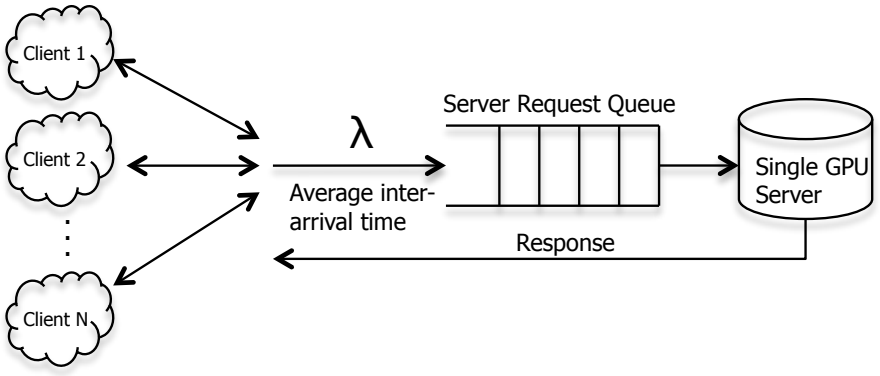

Figure 15: GPGPU application service model following a negative exponential distribution of request arrival from multiple end users.

- ***Memory Bandwidth Feedback (MBF):*** the MBF policy uses as input from device-level scheduling the approximate memory bandwidth of an application, by taking the ratio of the total data accesses by its computation kernels to the total time spent on the GPU. Workload balancing uses this information to avoid collocating bandwidth-bound threads. Resulting performance improvements leverage the fact that GPU-resident non-bandwidth/compute bound threads can hide the memory latencies experienced by bandwidth-bound GPU kernels.

2.4.4 Discussion

Important and novel about the GPU scheduling policies described above (vs. those targeting CPUs) is the explicit attention paid to data movement to/from GPUs. Phase Selection (PS) attempts to schedule requests differing in the phases in which they operate: computing vs. moving data. Feedback not only concerns GPU execution, but also to distinguish compute- from more movement-intensive GPU requests, refined by one specific quantification of memory movement: the approximate level of memory bandwidth consumed by GPU requests. While the implementation of these functionalities in the current Strings system is for CUDA devices, they are equally important for other accelerators, including those integrated into the platform (e.g., AMDs Fusion architecture) and those using alternative accelerator architectures (e.g., Intel's Xeon Phi).

Table 1: Benchmark Applications

Program	*Long-running Jobs (Group A)*			
	Input	**GPU Time (in %)**	**Data Transfer (in %)**	**Memory Bandwidth (in MB/s)**
DXTC (DC)	512 x 512 pixels	89.31	0.005	63.14
Scan (SC)	1K & 256K elements	10.73	24.99	1193.03
Binomial options (BO)	1024 points; 2048 steps	41.06	98.88	3764.44
Matrix multiply (MM)	480 x 480 elements	80.13	0.01	2143.26
Histogram (HI)	64-bin & 256-bin	86.51	0.17	13736.33
Eigenvalues (EV)	8192 x 8192 elements	41.92	0.73	401.27
	Short-running Jobs (Group B)			
Blackscholes (BS)	8000000 points; 1024 steps	24.51	6.23	50.23
MonteCarlo (MC)	2048 points	84.86	98.94	3047.32
Gaussian (GA)	50 x 50 elements	1.14	0.32	17.89
Sorting Networks (SN)	1M elements	2.05	26.68	320.35

2.5 Experimental Evaluation

The purpose of the experimental evaluations shown below is twofold. First, they show the importance of explicit accelerator scheduling for the cloud and server workloads seen in future datacenter systems. Second, device-level feedback about application characteristics and behavior is shown to be critical for obtaining high throughput and efficiently utilizing accelerator resources.

2.5.1 Evaluation Metrics

We use weighted speedup [106] and Jain's fairness [65] as metrics to measure overall system throughput and fairness, respectively. Weighted speedup measures the average speedup in an application when running alone compared to when the application is sharing the GPU. The fairness metric measures per-application fairness achieved when two or more

Table 2: Mapping from Workload Mix Label to Application Pair

A	DC, BS	**G**	SC, GA	**M**	MM, BS	**S**	HI, GA
B	DC, MC	**H**	SC, SN	**N**	MM, MC	**T**	HI, SN
C	DC, GA	**I**	BO, BS	**O**	MM, GA	**U**	EV, BS
D	DC, SN	**J**	BO, MC	**P**	MM, SN	**V**	EV, MC
E	SC, BS	**K**	BO, GA	**Q**	HI, BS	**W**	EV, GA
F	SC, MC	**L**	BO, SN	**R**	HI, MC	**X**	EV, SN

applications share the GPU where each is allocated a pre-defined share of the resource.

$$Weighted\ Speedup = \frac{1}{N} * \sum_{i=1}^{N} \frac{T_i^{alone}}{T_i^{shared}} \tag{2}$$

$$Jain's\ Fairness = \frac{(\sum_{i=1}^{N} \frac{T_i}{W_i})^2}{N * \sum_{i=1}^{N} (\frac{T_i}{W_i})^2} \tag{3}$$

2.5.2 Benchmarks

Applications from the CUDA SDK and the Rodinia benchmark suite [17], listed in Table 1, are chosen to create a pairwise workload mix of short (< 10sec) and relatively long (10-55 sec) running jobs. 24 such workload pairs shown in Table 2 are used, labeled from A to X, where A is the DC-BS pair, B is the DC-MC pair, X is the EV-SN pair, and so on, following the order in Table 1. We assume typical cloud services to operate in response to end user requests, with each individual service running for some small amount of time to complete a single request, but with the requirement of being highly responsive. This assumption matches the service behavior reported by Amazon, for instance, for its web service infrastructure. However, the actual types of service instances being used will vary, which we reflect by carefully choosing for the evaluation a diverse set of application kernels, like image processing (e.g., matrixmult), financial (e.g., BlackScholes), etc. The outcome is a workload mix with many short running rather than a few long running sets of jobs.

2.5.3 Experimental Setup

Experiments are performed on two different classes of servers, a small-scale (two GPUs) server and a higher end (four GPUs) server emulated by a supernode comprised of two dedicated dual-GPU nodes (NodeA and NodeB) connected via Gigabit Ethernet. Each of

the two machines is equipped with two Intel Xeon X5660 processors running at 2.8 GHz, for a total of 12 cores and 12 GB of RAM, and has two attached NVIDIA FERMI GPUs. NodeA has a Quadro 2000 and Tesla C2050, while NodeB has Quadro 4000 and Tesla C2070 GPUs, resulting in a supernode with a heterogeneous GPU pool where GPUs differ in terms of their compute and memory bandwidth capacities. The CUDA runtime and driver versions are 5.0 and 319.49, respectively. Our GPGPU application service model, as shown in Figure 15, is based on the SPECpower_ssj2008 benchmark [5], which models a server application with a large number of end users. User requests follow a negative exponential distribution and are served by a finite number of server threads. The exponential distribution models intermittent periods of bursts of load when application requests queue up while other requests are being processed, followed by periods of calm when the accumulated requests are serviced. For a particular random stream of requests, the inter-arrival time between any two consecutive requests, can be calculated using the following formula:

$$T = -\lambda * log(X) \tag{4}$$

where λ is the mean inter-arrival time between consecutive requests, and X is a random number in the range (0.0, 1.0].

NodeA and NodeB are servers processing GPU application requests. In our small-scale server experiments, NodeA sees a stream of requests following a negative exponential distribution, as described above, with proportional to the applications runtime. In the emulated high-end server experiments, each node sees independent random streams of requests. In both the small and large-scale server experiments the assumption is is large enough to handle the memory pressure on GPUs being shared, in other words GPU requests never pile up to the degree that they run out of device memory.

2.5.4 Results

Importance of Workload Balancing. In this set of experiments with a small-scale server, a node receives a stream of requests for a particular application following a negative exponential distribution. As shown in Figure 16, the average completion time of all requests served is calculated and compared with the different workload balancing policies of Rain

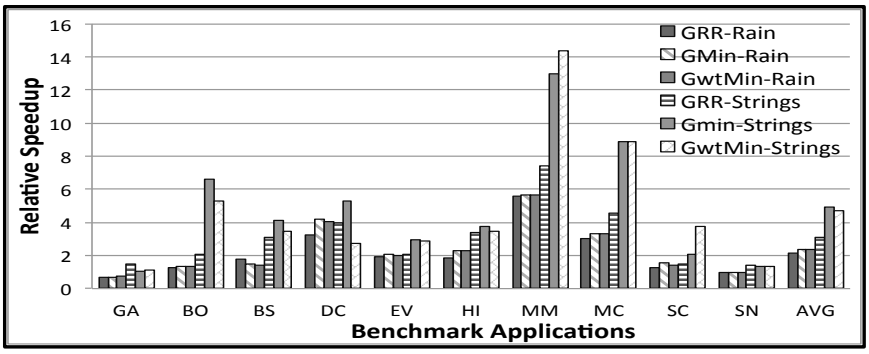

Figure 16: Performance benefit of workload balancing policies vs. CUDA runtime in a single node with 2 GPUs.

and Strings with the baseline being CUDA runtime (relative speedup). We can observe that there is significant throughput improvement with both our former solutions, Rain, and also with Strings, because unlike the CUDA runtime, which respects the applications target GPU selection, the workload balancer ignores it and dynamically distributes GPU requests across all GPUs in a node. This keeps both of the GPUs in the node busy, avoiding static GPU request collisions, thereby increasing the overall GPU utilization and achieving high system throughput. We also observe that every Strings workload balancing policy performs better than its Rain counterpart. This is because applications mapped to any GPU by Strings belong to the same GPU context and are dispatched over independent CUDA streams. This promotes concurrent execution (1) via time and space sharing the GPU and (2) by keeping both compute and memory copy engines of the GPU busy simultaneously. Averaged over all benchmark applications, the GRR-Rain, GMin-Rain, GWtMin-Rain, GRR-Strings, GMin-Strings and GWtMin-Srings policies achieve weighted speedups of 2.16x, 2.37x, 2.34x, 3.10x, 4.90x, and 4.73x, respectively, compared to the CUDA runtime. Further, on average, Strings workload balancing performs up to 2.10x better than Rain. Interestingly, for some applications (BO, BS and DC), note that GMin outperforms GWtMin, although the latter is, in principle, a better scheduling policy. This is because the static GPU weights assigned to each GPU during system initialization, in many cases, do not mirror the actual relative differences in application performance and therefore, fail to account for diverse application characteristics. This is a key motivation for feedback-based load balancing evaluated later.

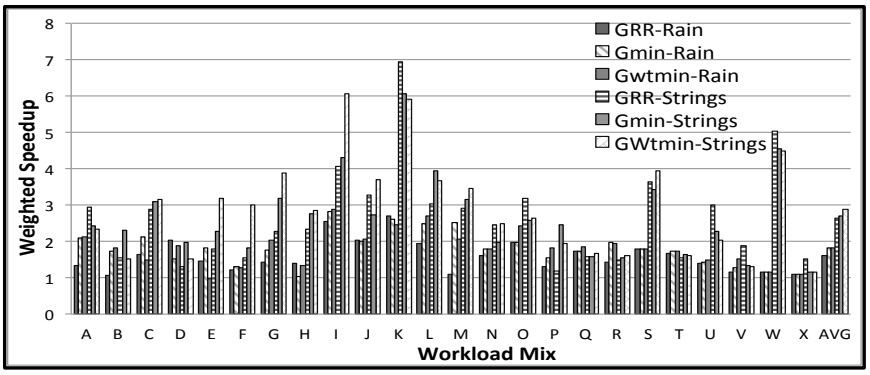

Figure 17: Performance benefit of GPU sharing in an emulated 4 GPU server.

In Strings, for the workloads Gaussian and Scan, GRR performs better than GMin, which seems counter-intuitive as GMin in theory should perform better than GRR (GMin takes into account the differences in application runtimes by maintaining a record of the number of applications currently bound to a particular GPU). This exception happens because unlike Rain, for Strings the per-GPU request queue length (used in the logic of the GMin policy) does not exactly capture the current load in the device. In Rain, the product of queue length and average application runtime in the queue is the total time for servicing the entire queue, but this is not true in Strings, with its multiple GPU requests serviced concurrently.

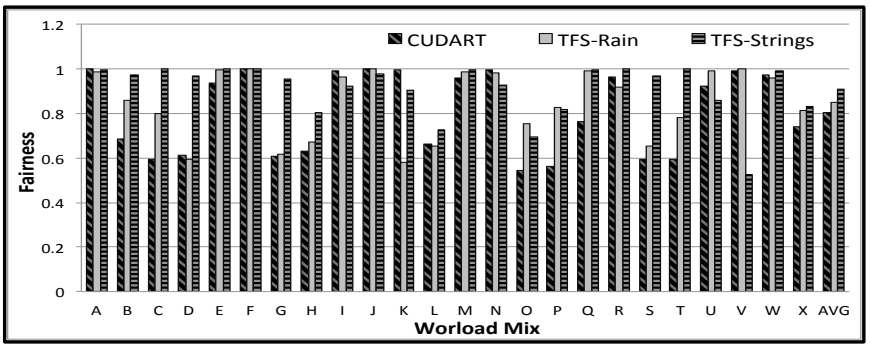

Figure 18: Fairness achieved by TFS-Strings vs. TFS-Rain vs. CUDA runtime.

Benefits of GPU Sharing. Experiments with the emulated (two node) larger-scale server machine show the benefits of sharing GPUs among multiple application streams. In these experiments, one node receives a stream of longer running GPU requests (Group A) whereas the other receives a stream of shorter requests (Group B), for two different

applications and following a negative exponential distribution. Leveraging the gPool, for these sets of requests, the workload balancer dynamically distributes them across all four GPUs in the supernode. Experiments record the average completion time of each application for each of the different workload balancing policies, with the baseline being the single node GRR policy. This means that the performance benefits reported are over and above those obtained with the single node GRR policy described in the previous section. As shown in Figure 17, averaged over 24 workload pairs, taking one each from Group A and B, the speedups achieved by GRR-Rain, GMin-Rain, GWtMin-Rain, GRR-Strings, GMin-Strings, and GWtMin-Strings policies are 1.60x, 1.80x, 1.82x, 2.64x, 2.69x and 2.88x respectively. These speed ups include the effects of both GPU sharing and workload balancing. These notable speedups are derived in part from the fact that the peaks in GPU requests from the two statistically independent streams are not aligned, thus allowing the workload balancer to distribute GPU requests efficiently among all four GPUs. We also observe that for all of the application pairs, maximum speedups are achieved in workloads (I, K and W) in which one of the applications is either BlackScholes or Gaussian. This is because for both of these benchmarks, relative GPU usage is among the lowest, thus benefiting the other application in the workload mix. Namely, BlackScholes has the least total execution time while the Gaussian kernel has very low GPU utilization, minimal data transfer, and negligible memory bandwidth. Finally, Strings outperforms Rain in GPU sharing because (i) the effect of GPU sharing becomes even more prominent with the opportunity of concurrent execution of GPU requests, and (ii) as shown in the motivation section, a consolidated single GPU context per device makes GPU usage much more uniform, allowing the efficient collocation of multiple requests.

Benefits of GPU Scheduling. We next evaluate a fairshare (TFS) and two throughput-oriented (LAS and PS) GPU scheduling policies.

1) Fairshare Scheduler: in this set of single node experi- ments, application pairs share a single GPU, each assigned equal GPU shares. From Figure 18, we see that TFS-Strings outperforms both the CUDA runtime and the TFS-Rain scheduling policy. The average fairness achieved by TFS- Strings is 91%, which is 13% and 7.14% better than the CUDA

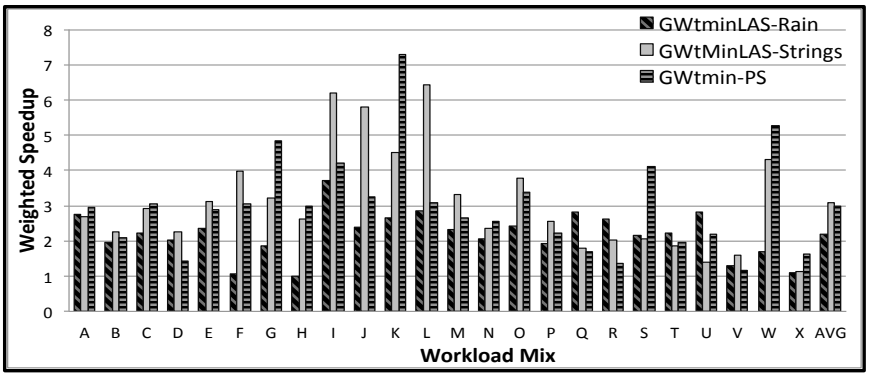

Figure 19: Performance benefit of GPU scheduling.

runtime and TFS-Rain, respectively. The maximum fairness achieved by TFS-Strings is close to ideal (99.99%), for the following reasons. By keeping a history of the GPU time attained by individual applications and penalizing any application in subsequent epochs that used the GPU for more than its allocated share in a previous epoch, both TFS-Rain and TFS-Strings achieve better fairness compared to the CUDA runtime. TFS-Strings achieving higher fairness compared to TFS-Rain might seem counter-intuitive, because the concurrently executing GPU requests from different applications in Strings makes it difficult to track or control fairness in GPU allocation. Better fairness in TFS-Strings can be explained by the fact that because there is no GPU context-switching among applications sharing a GPU in Strings, the error in the calculation of GPU usage of an application in a particular epoch is substantially reduced compared to that of Rain where the GPU context-switching overhead is included in the fairness calculation along with the applications actual GPU usage.

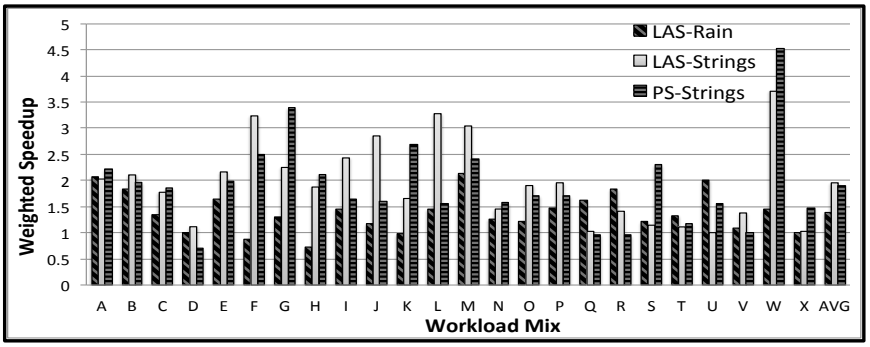

Figure 20: Performance benefit of GPU scheduling policies.

2) Throughput Oriented Scheduler: the baseline for this set of experiments is again the single node GRR policy, and scheduling policies are evaluated in combination with the best performing workload balancing policy from GPU sharing, i.e. GWtMin. As shown in Figure 19, the average weighted speedup achieved with GWtMinLAS-Rain, GWtMinLAS-Strings, and GWtMin-PS is 2.18x, 3.10x, and 2.97x, respectively. The higher speedup achieved by LAS is because it greedily prioritizes the GPU requests that have shorter GPU episodes and have consumed less GPU time until the most recently completed scheduling epoch. This helps in completing the short running GPU jobs faster and thus increasing the overall system throughput. PS is a Strings-only scheduling policy specifically designed to leverage the concurrency opportunity exposed by CUDA streams and tries to keep all the hardware units in a GPU busy by favoring applications which are in different phases of their use of the GPU, to be active simultaneously. Although it slightly falls short in comparison to GWtMinLAS-Strings, by just 4%, it does significantly better than GWtMinLAS-Rain, by almost 27%. Therefore, PS achieves nearly the same throughput as LAS but is not as unfair as LAS, which is an extremely greedy policy by definition and thus, might starve applications with relatively longer GPU episodes. The performance benefits shown in Figure 19 include the effect of both GPU sharing and device level GPU scheduling. To depict solely the benefits of GPU scheduling, Figure 20 compares the GPU scheduling policies with the baseline of GRR policy with four GPUs shared. LAS-Rain, LAS-Strings and PS-Strings achieve 1.40x, 1.95x and 1.90x throughput improvement respectively over this baseline.

Importance of Device-level Feedback. Feedback-based policies are evaluated relative to the single node GRR policy. When the workload balancer receives feedback information from low-level GPU schedulers, it dynamically switches to the appropriate feedback-based load balancing policy. Figure 21 shows the weighted speedup achieved by two feedback-based policies, RTF and GUF, for both Rain and Strings. Average speedups attained are 2.22x, 2.51x, 3.23x, and 3.96x for RTF-Rain, GUF-Rain, RTF-Strings, and GUF-Strings, respectively. Compared to the highest speedup achieved by the previously discussed non-feedback based policy (GWtMinLAS-Strings), RTF-Strings and GUF-Strings achieve 4.5% and 28% improvements, respectively. Feedback-based policies outperform

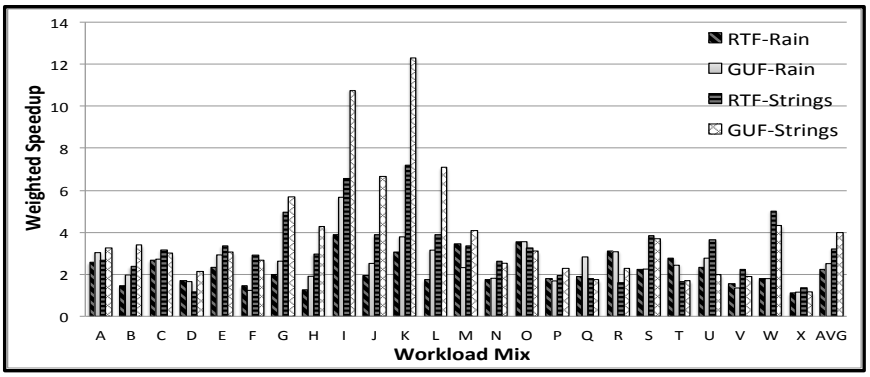

Figure 21: Performance benefit of feedback-based load balancing.

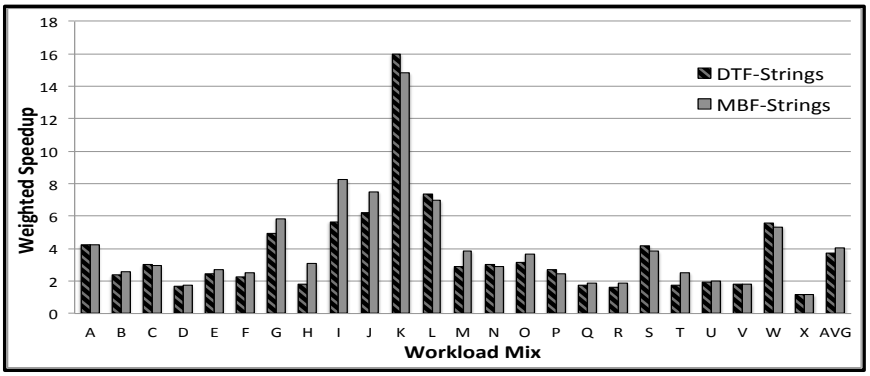

Figure 22: Performance benefit of two Strings specific feedback-based load balancing policies.

static workload balancing, because they make use of more detailed information about application characteristics. Unlike GWtMin, which is a proactive scheduling policy that relies on static weights assigned to GPUs, RTF employs a reactive scheduling technique that uses the actual GPU-specific runtimes of applications to balance load. GUF outperforms RTF in the workload mix of applications that have widely contrasting GPU utilization, i.e., pairs with very high (DC, HT, MM and BO in Group A) and very low (Gaussian, SN and BS in Group B) GPU utilization, by not collocating multiple applications with high GPU utilization on the same GPU.

Finally, we evaluate two Strings-specific feedback policies, DTF and MBF, which are designed to exploit the advantages offered by CUDA streams and context packing. As shown in Figure 22, DTF and MBF achieve average speedups of 3.73x and 4.02x, respectively. DTF performs best in a workload mix of applications with contrasting degrees of data transfer

demands and GPU compute times, i.e., application pairs that have high GPU compute times (DC, EI, HT and MM in Group A) and high data transfer times (MC and SN in Group B). So, when one application is busy in data transfer to the device, the other can continue to use the device for computation. MBF, which is the best performing feedback-based policy, optimally performs in the workload mix of applications with sufficient asymmetry in their memory access behavior, i.e., application pairs with high GPU compute time but low memory bandwidth (EV and DC in Group A) and with high memory bandwidth (BS, HI and MC in Group B). This allows a higher degree of parallelism, by hiding the long memory latencies of memory-bound applications via a runtime switch to a compute-bound application. It is also important to note that by definition, MBF includes the benefits of both RTF and DTF, because the methodology to calculate the approximate memory bandwidth (the ratio of the total data accesses by its computation kernels to the total time spent in the GPU) includes both the data transfer and running time information of the application. This makes MBF perform better than RTF and GUF, across almost all workload mixes. MBF performs better than GWtMinLAS-Strings, RTF, and GUF by more than 30%, 24%, and 1.5% respectively. Therefore, the maximum weighted speedup achieved across all the policies is 4.02x (MBF) relative to single node GRR policy, and 8.70x compared to the bare CUDA runtime.

Discussion. Key insights from the experimental results in this section are the following. (1) GPU underutilization due to static collisions is avoided by making GPUs into explicitly scheduled entities, thus enabling the load balancing of GPU requests over an aggregated shared GPU pool (gPool). The outcome is an average speedup of up to 4.90x compared to the CUDA runtime. (2) GPU core idling can be reduced further by packing the GPU components of applications sharing a GPU into a single GPU context, causing improvements over schedulers without this ability of an average 2.10x for representative server workloads. (3) Fine-grain feedback from device-level scheduling to load balancing is important, as demonstrated by the fact that GMin performs better than GWtMin for some applications, despite the latter policys theoretical superiority. More generally, feedback policies outperform other workload balancing methods, achieving an average speedup

of 8.70x. This is because feedback policies take into account fine grain application characteristics in their decision making and can thus maximize GPU utilization by collocating applications with contrasting behavior in terms of data transfer and memory intensity. (4) Substantial advantages can be derived from collocating on the same device, streams of requests with likely unaligned peak performance demands, resulting in more uniform GPU usage patterns. Such uniformity is further encouraged by context packing and the consequent sharing of GPU context. (5) An important outcome of this work is that scheduling must go beyond considering GPU resources to also consider other schedulable device components, i.e., the GPUs data movement engines. This is shown by the phase selection policy leveraging CUDA streams to keep all hardware units in a GPU busy. It achieves system throughput of 6.41x over the CUDA runtime, and it performs as well as the greedy, unfair LAS policy, but without compromising fairness.

2.6 Related Work

From the existing body of work on heterogeneous schedulers, StarPU [29] and Symphony [87] employ a dynamically learned performance model to decide which of the available resources to use, assuming optimized implementations of the same application targeted to different resources exist and can be dynamically dispatched to any one of them as needed. Previous work on interference-driven GPU resource management [85] aims to provide better GPU utilization in heterogeneous clusters, by co-locating multiple jobs to share same GPU, respecting GPU memory constraints, but unlike Strings, which decouples the CPU and GPU components of a job and schedules them separately, [85] schedules both application components together on the same node, which might be restrictive. Further, Strings dynamically learns the applications GPU characteristics, whereas [85] performs static profiling.

Recent work [32] on managing GPU memory pressure arising from consolidating multiple applications on a single GPU is an interesting complement to our work. Similar middleware infrastructures have recently been proposed for other heterogeneous architectures like Intel's Xeon Phi [37]. By incorporating the virtual memory support of [37, 32], Strings can eliminate the assumption on the maximum rate of request arrivals in GPU servers. Gdev [66]

implements open source versions of CUDA driver and the runtime library, that builds virtual memory support for GPUs inside the OS kernel unlike [16] which presents its virtual memory abstraction at the runtime API level. With the Gdev open source driver, both the context packer module and device-level scheduler of Strings can be pushed inside the OS kernel, eliminating runtime layer overheads. GEMTC middleware infrastructure [70] implements a dynamic memory management system that efficiently allocates memory on the GPU. But unlike Strings, which is completely transparent to the applications, GEMTC exposes a set of new memory management APIs that require applications to be rewritten. Moreover, Strings supports management of heterogeneous multi-GPU resources on a single node which GEMTC infrastructure is yet to support.

In general, previous work on GPU virtualization GVim [59], Pegasus [60], vCuda [102], rCuda [44], gVirtus [55] make the GPUs visible from within the virtual machine. GVim and Pegasus both do QoS-aware scheduling by using a working queue per GPU, but they do not address the problem of scheduling GPU requests across multiple GPUs within a node or across multiple nodes in a cluster. Pegasus also explores the co-scheduling of GPU request with corresponding VCPUs. This can be combined with our Strings infrastructure by gang scheduling decoupled application's CPU and GPU components. vCuda and rCuda leverage the multiplexing mechanism of the bare CUDA runtime for GPU sharing, but they don't look into scheduling policies targeting various resource management goals.

2.7 Chapter Summary

This chapter presents the Strings scheduler and scheduling policies for GPUs as first class schedulable entities in high-end cloud services. Decomposing the scheduling problem into a combination of workload balancing and device-level scheduling, Strings contributes scheduling policies that explicitly consider data movement to/from accelerators, methods that dynamically encapsulate the GPU contexts of multiple applications into a single umbrella context to achieve high GPU utilization and minimize context switching overhead, and support that makes possible the runtime switching of policies based on device-level scheduler feedback. Novel scheduling policies developed with Strings include (i) a Phase Selection

(PS) policy that aims to keep all of a GPUs hardware units busy by smartly picking and simultaneously running applications operating in different phases of their use of the GPU, (ii) advanced feedback-based policies like DTF and MBF that exploit the advantages offered by CUDA streams and context packing, by collocating applications with contrasting behavior, in terms of data transfer and memory intensity, to achieve extreme performance benefits, (iii) a throughput oriented greedy but highly unfair LAS policy favoring jobs with least-attained levels of GPU service, and (iv) history-based fairshare scheduling that improved fairness in GPU usage for multi-tenant applications.

Extensive experimental evaluations across a wide variety of workloads and system configurations shows Strings to achieve speedups of up to **4.90x** and **2.07x** on a single node compared to the CUDA runtime and over our own previous GPU scheduling work, Rain, respectively. Averaged over 24 pairs of short and long running workload mix, Strings achieve a weighted speedup of up to **8.70x** and **13%** improvements in fairness over the CUDA runtime.

Our future work will consider dynamic opportunities and tradeoffs in mapping executions to either GPUs or CPUs, using runtime methods for binary translation [42]. Also interesting is further exploration of the effects of data movement on program performance and consequent changes in scheduling policies, first for discrete vs. integrated GPUs and second considering GPU multi-tenancy.

CHAPTER III

GRAPHREDUCE: PROCESSING LARGE-SCALE GRAPHS ON ACCELERATOR-BASED SYSTEMS

The need to rapidly process large graph-structured data, in both scientific and commercial applications, has engendered recent efforts to leverage cost-efficient GPUs [96, 97] for efficient graph analytics. Doing so, however, requires addressing substantial technical challenges, including (1) dealing with the dynamic nature of graph parallelism [119, 52, 67, 51], (2) coping with limited on-GPU memory, i.e., to process graphs with memory footprints that exceed limited GPU memory sizes [71, 93], and (3) addressing programmability issues for developers with limited insights into how to best exploit the resources of evolving and varied GPU architectures [79, 64, 74].

Previous work on parallel graph processing has sought to exploit scale-out methods, by distributing large graph data across the different nodes of computational clusters [77]. Recognizing the low computation to communication ratios of typical graph processing algorithms [71, 93], the 'GraphReduce' (GR) programming framework presented in this chapter uses the alternative 'scale up' approach in which large graphs processed by memory-limited GPUs can take advantage of the potentially considerable memory capacities of the host machines to which they are attached. The implementation of GR for NVIDIA GPUs evaluated in this chapter efficiently runs irregular graph algorithms on datasets considerably larger than GPU memory sizes, by (i) partitioning graphs into fixed size chunks – shards – asynchronously moved between GPU and host, (ii) adopting a combination of of edge-(X-Stream [93]) and vertex-(GraphChi [71]) centric implementations of graph representations, (iii) overlapping GPU computation with data transfer via concurrent GPU operations, using CUDA streams, and (iv) using 'spray' operations to further divide shards and obtain fine-grain parallelism that exploits the Hyper-Q feature of Kepler GPUs [13]. Specifically, spray operations are used to further divide each shard into multiple sub-buffers transferred

over dynamically created CUDA streams. The purpose is efficiently use GPU hardware features like Hyper-Qs.

GraphReduce runs graph algorithms on GPUs without unduly burdening graph algorithm developers. Programmers write the appropriate sequential codes for their algorithms, e.g., for data mining, machine learning, etc., and then use its simple API to express their use for processing entire graphs. The GR runtime partitions the graph into different shards, each single one of which entirely fits into GPU memory. Graph processing, then, overlaps shard movement with GPU-level graph processing, the latter using multiple levels of GPU-level parallelism, as indicated above. With such automation, GR can deal with graph sizes much exceeding GPU memory sizes. This is important because even a common Yahoo web-graph comprised of 1.4 billion vertices [25] requires approximately 6.6 GB of memory to store just its vertex values (not even including the edges and their status).

In summary, with GraphReduce, GPUs can be used to accelerate analytics performed on graphs with billions of edges, operating at speeds much exceeding that of similar operations run on CPUs, and programmed in ways accessible to programmers who are not experts in GPU programming. To the best of our knowledge, GraphReduce is the first to support in-GPU-memory and out-of-GPU-memory graph processing, thus aiming for scale-up graph processing on HPC systems with discrete GPUs and high end (i.e., memory-rich) hosts.

The GraphReduce graph processing framework uses a Gather-Apply-Scatter (GAS) model to efficiently process graphs of sizes larger than GPU memory. Its technical contributions include the following:

- High performance is obtained in part from its use of a combination of edge- and vertex-centric graph programming, to match the different types of parallelism present in different phases of the GAS execution model.

- Efficiency in graph processing via improved asynchrony in computation and communication, gained by GR's runtime via dynamic characterization of data buffers based on data access pattern and access locality. Additional hardware parallelism is extracted via spray streams for deep copy operations on separate CUDA streams.

- Use of computational frontier information for efficient GPU hardware thread scheduling and data movement between host and GPU. Specifically, GR moves data into GPU memory only when a subset of the graph has at least one active vertex or edge. Further, when possible, GR uses dynamic phase fusion/elimination to merge/eliminate multiple GAS phases, to avoid unnecessary kernel launches and associated data movement.

Results show that GraphReduce can significantly outperforms the sate-of-the-art graph analytics frameworks across a wide variety of algorithms, for large-scale graphs that do not fit into GPU-resident memory. Specifically, it achieves up to **79x and 21x**, and an average of **13.4x and 5x** speedup over GraphChi [71] and X-Stream [93] respectively, for several real-world large-scale graphs with various algorithms. Additionally, GraphReduce also achieves comparable performance with the existing highly optimized in-GPU-memory solutions such as MapGraph [52] and CuSha [67], for smaller in-memory graph inputs.

The remainder of the chapter is organized as follows: Section 3.1 discusses the background and motivation for GraphReduce. Section 3.2 dissects the design choices. Section 3.3 introduces our GraphReduce framework. Section 3.4 highlights the major optimizations used in GraphReduce. Section 3.5 presents the experimental setup and result analysis followed by conclusion with future work.

3.1 Background and Motivation

This section introduces the computational model used in GraphReduce. It also motivates the GR approach by describing some of the challenges faced by the existing state-of-the-art graph processing approaches.

3.1.1 Computational Model: GAS Abstraction

GraphReduce exposes the Gather-Apply-Scatter (GAS) computational model used by Pregel [80], Powergraph [56], and GraphLab [77]. With GAS, a problem is described as a directed (sparse) graph, $G = (V, E)$, where V denotes the vertex set and E denotes the directed edge set. A value is associated with each vertex $v \in V$, and each directed edge e_{ij} is associated

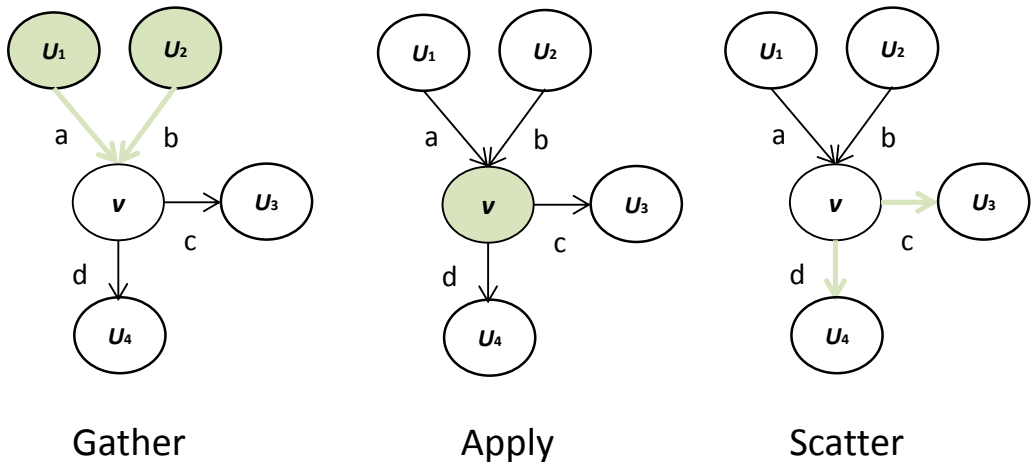

Figure 23: An example of GAS abstraction.

with a source vertex u and a target vertex v: $e_{ij} = (u, v) \in E$. Given a directed edge $e_{ij} = (u, v)$, we refer to e_{ij} as vertex v's in-edge, and as vertex u's out-edge. A typical GAS computation, then, has three stages [23]: (1) Initialization, (2) Iterations, and (3) Output. Initialization deals with initializing vertex/edge values and a starting *computation frontier*, which is defined as the set of active vertices for a given iteration. In each Iteration stage, a sequence of iterations is run, each gathering the values seen on the incoming edges, updating the values of elements, and then defining a new frontier for the next iteration. Figure 23 illustrates these three phases, assuming vertex v to be the central vertex.

- **Gather Phase**: each vertex aggregates values associated with its incoming edges and their source vertices. We define the gather function as $G(u, v, e_{ij})$, and we use binary operator $\biguplus$ to aggregate the outputs from multiple Gs into one value R. In Figure 1 (a), the result (R) from the Gather Phase for vertex v can be represented as $R = G(u_1, v, a) \biguplus G(u_2, v, b)$.

- **Apply Phase**: the value of each vertex in the current frontier is updated through the gather result. We define the update function as $U(v, R)$, where R is the result from the Gather Phase. Shown in the Figure 1 (b), we have the updated vertex v as: $v' = U(v, R)$.

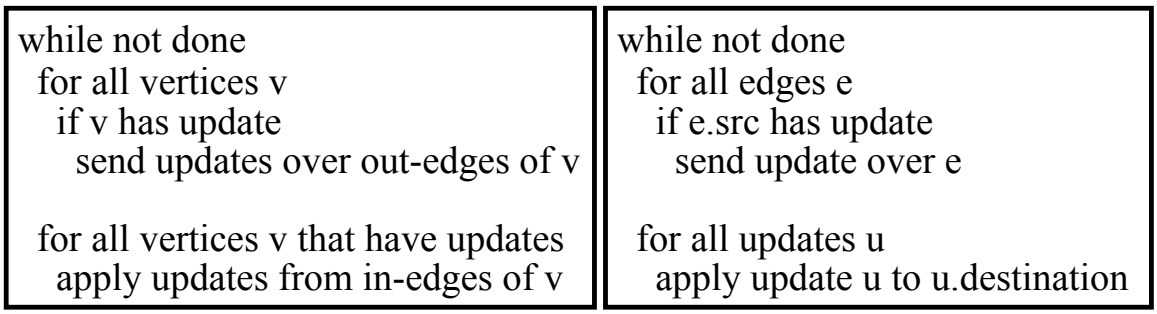

Figure 24: (a) Vertex-centric Scatter-Gather. (b) Edge-centric Scatter-Gather.

- **Scatter Phase**: the new vertex state is propagated to neighbors, by updating the state of its out-edges (e.g., c and d in Figure 1). We define the Scatter function for updating the out-edges of v as $S(v', e_{out})$, where v' is the updated vertex v and e_{out} represents v's out-edges. Shown in Figure 1 (c), two updated edges c' and d' are denoted as: $c' = S(v', c)$ and $d' = S(v', d)$.

As shown in much prior work [56, 80, 118], the GAS model is not only simple to use, but it is also sufficiently general to express a broad set of graph algorithms, ranging from PageRank to Connected Components, and from Heat Simulation to Sparse Linear Algebra. For example, the PageRank algorithm [30] can be expressed as follows. In the **Gather Phase**, each vertex v_i in the current frontier accumulates $G_i = \sum \frac{R_j}{n_j}$ from all in-edges from source vertex v_j, where R_j is the rank of v_j and n_j is the number of out-edges $(v_j \rightarrow v_i)$ of v_j. Then, in the **Apply Phase**, vertex v_i updates its value using some common PageRank formula like $R_i = 0.85 + 0.15 \times G_i$. Since in PageRank, the values of out-edges of v_i will not change in the **Scatter Phase**, there are no operations for this phase.

Figure 24 shows two common ways to implement graph algorithms with GAS: edge-centric vs. vertex-centric execution, which differs in whether the Scatter and Gather phases iterate over and update edges or vertices (their pseudo codes are shown in Figure 24). Implementation can also vary in terms of Update functions, to be implemented as either Bulk-Synchronous Parallel (BSP) [112] for simplicity or via asynchronous execution, for faster convergence. In either case, the graph algorithm terminates when some application-specific condition is met, e.g., when no more changes in vertex and edge states beyond a

certain threshold.

3.1.2 Motivation and Challenges

Graph Name	Vertices	Edges	In-memory Size
GPU In-Memory			
ak2010[6]	45,292	108,549	7.9MB
coAuthorsDBLP[8]	299,067	977,676	69.5MB
kron_g500-logn20[92]	1,048,576	44,620,272	2.4GB
webbase-1M[116]	1,000,005	3,105,536	211.6MB
belgium_osm[7]	1,441,295	1,549,970	5.4MB
GPU Out-of-Memory			
kron_g500-logn21[92]	2,097,152	91,042,010	4.84GB
nlpkkt160[94]	8,345,600	221,172,512	11.9GB
uk-2002[20]	18,520,486	298,113,762	16.4GB
orkut[117]	3,072,441	117,185,083	6.2GB
cage15[3]	5,154,859	99,199,551	5.4GB

Table 3: Datasets used to evaluate GraphReduce framework. 'Out-of-memory' means that the input graphs cannot fit into the limited GPU memory. A commercial K20c GPU with a 4.8 GB global memory is used as an example to illustrate in-memory and out-of-memory cases.

Graphs	X-Stream (ms)	CuSha(ms)	Speedup
ak2010	215.155	7.75	28x
belgium_osm	2695.88	791.299	3x
coAuthorsDBLP	1275	11.553	110x
delaunay_n13	80.89	5.184	16x
kron_g500-logn20	46550.7	119.824	389x
webbase-1M	3909.12	13.515	290x

Table 4: Performance comparision between two state-of-the-art graph processing approaches. X-Stream runs on a 16 core Xeon E5-2670 CPU with 32GB memory. CuSha runs on a NVIDIA K20c Kepler GPU with 4.8 GB memory.

3.1.2.1 Why Graph Analytics Using GPUs ?

The high compute power and multi-level parallelism provided by the SIMT (Single Instruction Multiple Threads) architectures of GPGPUs [1] present opportunities for accelerating many graph algorithms [119, 67, 23, 52]. Table 3 shows various in-GPU-memory and out-of-core graph datasets used to evaluate GraphReduce framework. Table 4 shows

[1]Without specified mention, NVIDIA terminology will be used throughout the chapter to illustrate our work. However, the proposed methodology can be easily applied to a wide range of massively parallel architectures.

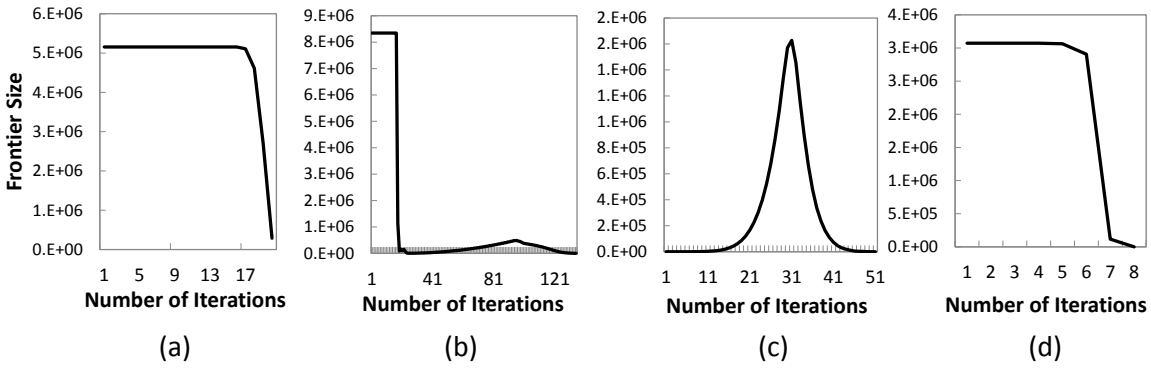

Figure 25: Frontier size changes across iterations using the GAS model on GPUs. This phenomenon highly depends on the input graph and algorithm, showcasing the inherent graph irregularity. Four cases from left to right: (a) Cage15 - PageRank; (b) nlpkkt160 - PageRank; (c) Cage15 - BFS; and (d) orkut - Connected Component (CC).

the performance comparison between two state-of-the-art graph analytics processing the BFS algorithm: X-Stream [93] for CPUs and CuSha [67] for in-memory GPU processing. Significant performance speedups are observed from using the GPU. For instance, graph kron_g500-logn20 [92] processed by CuSha on a commercial K20c Kepler GPU (4.8GB memory) achieves 390x speedup over X-Stream on a 16 core Xeon E5-2670 CPU (32 GB memory).

3.1.2.2 Challenges in GPU Graph Analytics

Acceleration of graph analytics via GPUs is limited, however, by the fact that many real-world graphs cannot fit into GPUs' limited memories. As mentioned earlier, a common Yahoo-web graph [25] with 1.4 billion vertices requires 6.6 GB memory just to store its vertex values. Additional examples of graphs exceeding GPU memory sizes appear in Table 3. Previous work on GPU-based graph processing has not addressed this issue. CuSha [67], MapGraph [52], VertexAPI [23] and Medusa [119] all assume graphs to reside in GPU memory. GraphChi [71] and X-Stream [93] are designed for CPU-based systems, unable to benefit from the multi-level massive parallelism offered by GPUs (shown in Table 4). Hybrid approaches (CPU+GPU) like Totem [53] are only able to process a fixed sub-graph that can fit into GPU memory after statically partitioning the graph between CPU and GPU, which results in underutilization of GPU's fullest processing power and parallelism.

There are several challenges to efficiently process larger-than-memory graphs on GPUs. They involve the need to provide end users with convenient programming constructs for their graph algorithms, but without unduly burdening them with (i) graph partitioning to fit sub-graphs into GPU memory, (ii) how and when such partitions are moved between GPU and host memories, and (iii) how to best extract multi-level parallelism from their GPU-resident execution. The GraphReduce framework presented in this chapter addresses these challenges.

Graph partitioning or chunking for fitting into GPU memory must deal with the irregular nature of graph algorithms and how they access the input data. More specifically, to obtain high on-GPU performance, chunking must be done to minimize GPU-host data movement. For the GAS model, this requires ensuring GPU memory residence of the vertices and edges that actively take part in the computation iterations being performed. This despite the fact that due to the inherent irregularity in graph algorithms, in every computation iteration, the number of edges and vertices that actively take part in computation (computation frontier size) is not constant, and it varies with graph algorithms and datasets, as shown in Figure 25. Across all of these cases, the frontier sizes [2] incur significant changes (either dropping or climbing). For instance, in Figure 25(b) for graph nlpkkt160 processed by PageRank, the frontier size drops sharply after a few iterations and remains low for the rest of the execution. Given these results, ideally, the GR runtime should *move sub-graphs to the GPU only if they contain active vertices and edges*. Otherwise, such movements simply cause unnecessary overhead. For the same nlpkkt160 case in Figure 25(b), after several iterations, most of the sub-graphs do not have active vertices/edges, so there is no need to move those chunks to the GPU. We have found this phenomena to be very common across most of the graphs shown in Table 3, for various algorithms. The GR methods presented in this chapter address this issue, along with (ii) and (iii) above.

[2]The term "frontier size" is synonymous with the number of active vertices in a given iteration. The variation of the frontier size during execution is sensitive to the starting-point for graph processing.

3.2 Design Choices

3.2.1 Hybrid Programming Model

Existing systems choose some specific programming model for graph execution. GraphLab [77], Pregel [80] and GraphChi [71] use the vertex-centric model, while X-Stream [93] uses the edge-centric model. In comparison, GraphReduce employs a hybrid programming model using both edge- and vertex-centric operations. This is because in the GAS model, different processing phases have different types of parallelism and consequently, offer different parallelism opportunities, coupled with different memory access characteristics. For instance, an edge-centric model should be used in the **Gather Phase** (shown in Figure 23), because a GPU hardware thread will then be assigned to work on behalf of an edge in the graph. This is preferable to the vertex-centric model, because first, real-world graphs commonly have more edges than vertices (shown in Table 3), thus giving rise to higher degrees of parallelism and decreased GPU core idling. Second, in the vertex-centric approach, each vertex receives information from multiple in-edges, resulting in a consequent need for synchronization or atomics to order the receive operations from each of the in-edges. This could potentially degrade the overall performance. The same observations hold for using the edge-centric model in the **Scatter Phase**. In contrast, in the **Apply Phase**, there are parallelism opportunities only over the vertex set, thus favoring a vertex-centric model.

3.2.2 Characterization of Buffers in Play

Graph data chunked to fit into GPU memory and to be moved from host to GPU, is comprised of edges that have either a destination or a source vertex in some well-defined graph partition (see Section 3.3.2 for details). Henceforth termed *shards*, such chunks reside in memory buffers that experience different access patterns. GraphReduce characterizes such access patterns in order to appropriately map corresponding memory buffers to the memory abstractions exposed by current GPUs. In terms of data movement, buffers can be classified as static vs. streaming buffers. Static buffers are copied only once to GPU memory, typically in the Initialization phase. They remain there for the lifetime of the graph execution. An example is a vertex set of a graph that fits into GPU memory. Streaming

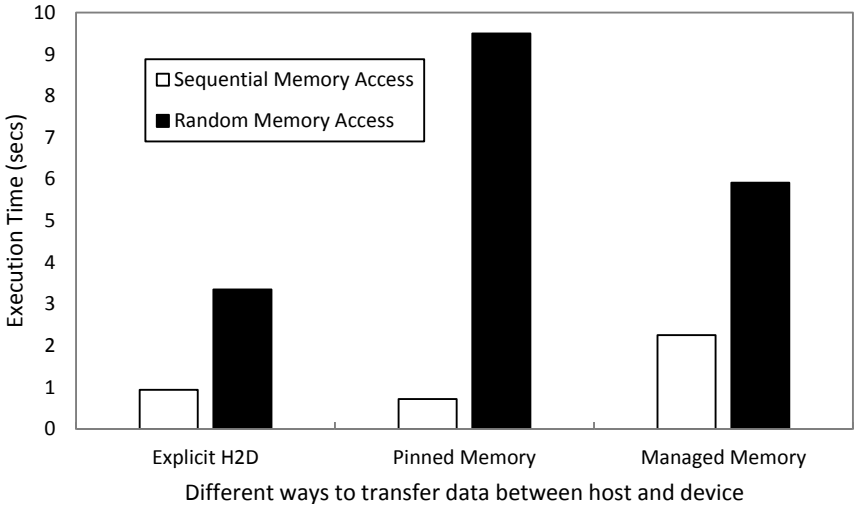

Figure 26: Performance of transferring 100 million double elements, using three techniques for data exchange between CPU and GPU.

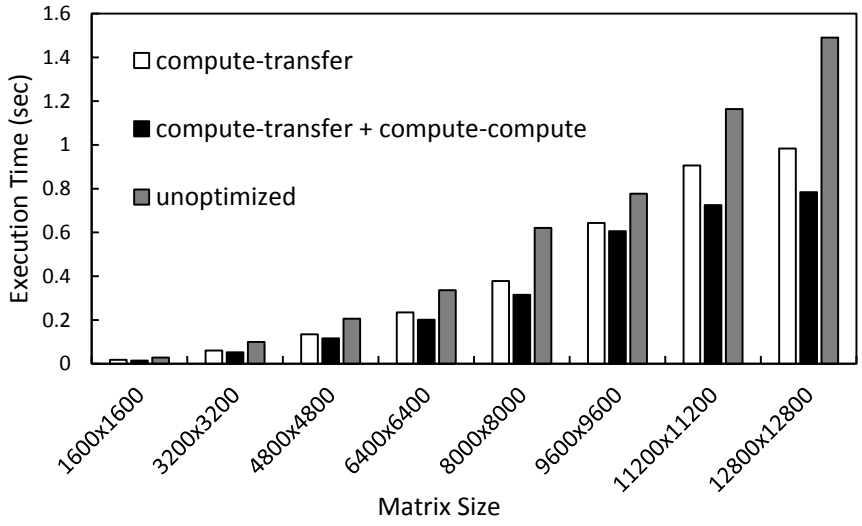

Figure 27: Performance benefits of using a combination of compute-transfer and compute-compute schemes for processing matrix multiplication with different input sizes. Stripe size=50, which refers to the contiguous number of rows of the matrix being fetched into the GPU memory as a chunk.

buffers, on the other hand, are moved in and out of GPU memory as processing progresses, and at any point in time, a particular instance of some streaming buffer resides in GPU memory, e.g., a subset of a graph's edge set. In the GAS programming model, static buffers are accessed in all three phases, while streaming buffers only appear in a single phase. Another way to characterize buffers is by their access rules, such as read-only or read/write access. For example, vertex and edge data buffers (containing mutable states) have both

read and write access patterns, while the vertex set (containing immutable vertex IDs) is read-only. Based on these attributes, the GR runtime makes decisions on whether or not to transfer certain buffers back to the host (see Section 3.3.3). Finally, buffers can be classified in terms of the spatial locality of their accesses, e.g., random or sequential access. For example, in the edge-centric approach shown in Figure 24, there are random accesses to the vertex set.

Once characterized, buffers are mapped to the different memory abstractions exposed by GPU, which at minimum, contain slow and fast memory [100] (e.g., host memory and GPU memory). In the different phases of the GAS model, there are a mix of random and sequential accesses to the input buffers (e.g., edge/vertex sets). For this mix, we posit that random access to slow memory is much more expensive than random access to faster memory, whereas for sequential access, memory-level parallelism and prefetching can help mask slower memory access speed. Therefore, due to the limited fast memory size (GPU), we choose to map all sequential accesses in a GAS phase to the slower CPU memory and all the random accesses to the faster GPU memory. We next validate these assumptions.

Figure 26 depicts the performance of three techniques for data exchange between host and GPU (through CUDA runtime APIs): (a) explicit data transfer using cudaMemcpy() or *Explicit H2D*; (b) *Pinned Memory* using Unified Virtual Addressing (UVA) [21], in which data is transferred implicitly by the CUDA runtime but the memory is allocated as locked memory on the host side; and (c) *Managed Memory* (introduced in CUDA 6 [21] as Unified memory), where data is transferred between host and device on demand. The measurements shown in the figure illustrate that in the case of sequential memory access, *Pinned Memory* performs the best, because the accesses directly translate to memory loads/stores operations over the PCIe in which (i) sequential accesses benefit from memory level parallelism (MLP) and (ii) software-level prefetching can hide communication overheads. In the case of random access, *Explicit H2D* performs the best and *Pinned Memory* performs the worst. In other words, random access performs best when data resides in faster GPU memory, and the performance of the *Pinned Memory* degrades as the load/store memory operations over the PCIe fail to benefit from prefetching (after all, accesses are random!). Since *Pinned Memory*

performs the best for sequential accesses, one straightforward approach is to organize graphs such that all memory accesses are sequential. However, because of the significant number of random accesses to either the edges or vertices of a graph in at least one phase of the GAS model [71, 93], this is not a viable solution for GR as the benefits of sequential accesses are overshadowed by the huge overhead of the random accesses to the slow memory. In response, GR uses explicit data transfer as the mechanism for transferring data between host and device, in way that aim to leverage GPU memory coalescing and software prefetching for the sequential accesses. Although certain performance benefits may exist through intellegent runtime buffer-type selecting. we leave this exploration for the future work.

3.2.3 Coordinated Computation and Data Movement

The spatial choice of where in memory to locate data requires an associated temporal choice in when to perform data movement between host and GPU memories. GR uses two methods to attain high performance: (1) hide communication costs by overlapping GPU computation with necessary data transfers, and (2) utilize the GPU's inherent high degree of potential internal parallelism. (1) is obtained via software-based prefetching to move shards into GPU memory while GPU kernel(s) are being executed. (2) is realized by leveraging underutilized GPU resources (idle threads) caused by the irregular nature of graph processing (shown in Figure 25). It involves (i) detecting such idle threads, using the computation frontier information available to the GR runtime, and (ii) initiating the execution of new shards when idleness is present (note that shards within a single GAS phase do not have data dependencies, so they can be processed in parallel). GR accomplishes this by automatically launching multiple kernels (within the same context), according to the resources available in each GAS phase. Denoting (1) as *compute-transfer* scheme and (2) as a *compute-compute* scheme, Figure 27 shows the performance benefits obtained from using these approaches vs. an unoptimized scenario when processing a large matrix that doesn't fit into GPU memory, thus clearly demonstrating the importance of coordinating computation with data movement. We will use these two schemes for processing graph algorithms across phases in the GAS model.

Connected Component (CC)

```
0.          __host__ __device__
1.          static int gatherReduce(const int& left, const int& right)
2.          {
3.               return min(left, right);
4.          }
5.          __host__ __device__
6.          static int gatherMap(const VertexData* dstLabel, const VertexData
                                 *srcLabel, const EdgeData* edge)
7.          {
8.               return *srcLabel;
10.         }
11.         __host__ __device__
12.         static bool apply(VertexData* curLabel, GatherResult label)
13.         {    bool changed = label < *curLabel;
14.              *curLabel = min(*curLabel, label);
15.              return changed;
16.         }
17.         __host__ __device__  static void scatter(const VertexData* src, const
                                                     VertexData *dst, EdgeData* edge)
18.         {
                //no scatter operations for CC algorithm
19.         }
```

Figure 28: Writing sequential code using GAS model for Connected Component (CC) algorithm in GraphReduce.

3.3 GraphReduce Framework

The GraphReduce framework can efficiently process graphs with large inputs and mutable edge values that cannot fit into the limited memories of discrete accelerators. GraphReduce simplifies graph analytics programming by supporting a multi-level, asynchronous model of computation. Figure 30 shows the general software architecture of GraphReduce which consists of three major components: Partition Engine, Data Movement Engine, and Compute Engine, all supporting an easily used GAS-based API.

3.3.1 User Interface

As shown in Figure 28, programmers can write a sequential algorithm by simply defining the graph's state data types (for vertices and edges) and four functions for the different phases in the GAS programming model. GraphReduce will then seamlessly generate parallel codes to run on the GPU. User-defined functions include $gatherMap()$, $gatherReduce()$, $apply()$ and $scatter()$, corresponding to the functions defined in Section 3.1.1, i.e., to $G()$, $\uplus$, $U()$,

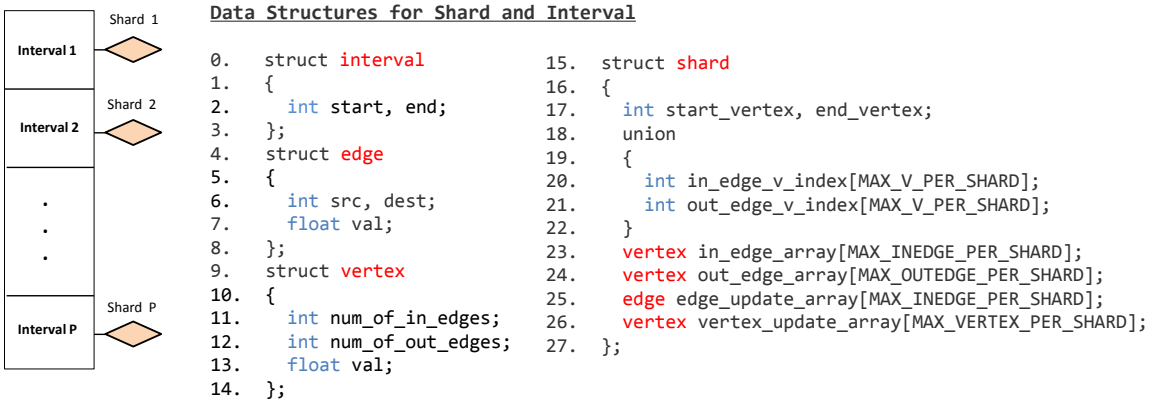

Figure 29: Illustration of *shard* and its data structure.

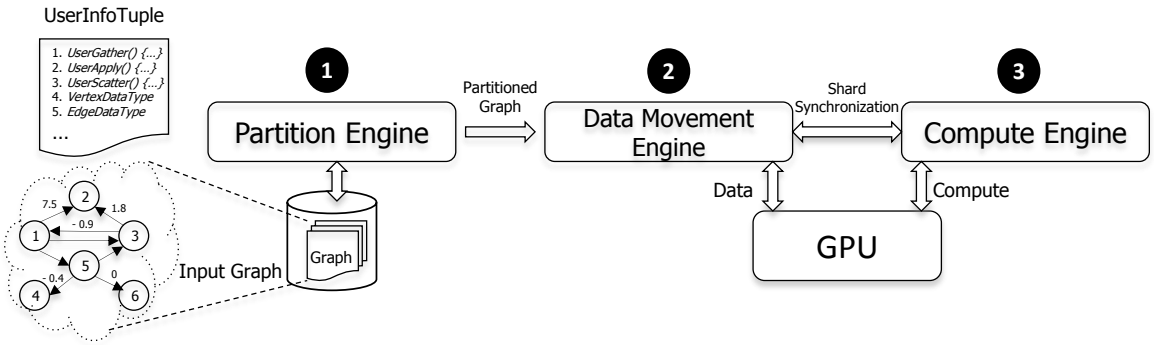

Figure 30: Architecture of GraphReduce framework.

and $S()$. Along with the vertex and edge data types, these functions are stored in a tuple called UserInfoTuple: $<gather()$, $apply()$, $scatter()$, $VertexDataType$, $EdgeDataType>$.

3.3.2 Partition Engine

The *Partition Engine* shown as **1** in Figure 31 is responsible for (1) load-balanced shard creation and (2) providing graph partitioning logics and associated orderings of vertices/edges. Partitioning is performed by dividing the vertex set V of graph $G = (V, E)$ into disjoint intervals (i.e., sets of vertices) and for each interval, maintaining a *shard* data structure (shown in Figure 29), where each shard stores all of the edges that have either a destination or a source vertex in that interval.

GraphReduce answers the following questions about such sharding: (1) choice of interval, (2) number and sizes of shards, and, (3) how to order the edges in each shard. For (1), shown in Figure 29, intervals are chosen by the Shard Creator of the Partition Engine in a

load-balanced fashion. Specifically, each shard contains an approximately equal number of edges (in- plus out-edges). For (2), the number of intervals P is chosen such that at least one shard (maybe multiple) can be loaded completely into GPU memory. Therefore, if more than one shard is allocated in GPU memory, the total number of shards simultaneously participating in graph computation can be calculated as a function of the total number of concurrent memory copy operations to and from the GPU. Finally, for (3), the graph dataset is a set of source and destination vertex pairs (edges) with the associated value for each edge. This set of tuples is generally unordered. The Graph Layout Engine inside of the Partition Engine defines the layout of the data by sorting the in-edges in the order of their destinations and the out-edges in the order of their sources. For such sorted data, we use the compressed Sparse Column (CSC) and compressed Sparse Row (CSR) formats [33] to store graphs to be used in the Gather and Scatter phases, respectively. Therefore, there is no overhead for runtime data-format transposition between CSC and CSR formats.

Edges are stored in some specific sorted order for three reasons. First, with ordered edges, the data moved across the PCIe link from host to GPU is contiguous, which can improve system throughput. Second, when updates are collected for each vertex in the gather phase, they can be stored in consecutive memory locations for each vertex, which avoids random memory accesses. Similarly, the out-edges are stored in the order of source vertices, so that the neighbors of a vertex whose states have been updated can be accessed sequentially. Third, sequential accesses to GPU memory can improve performance due to memory coalescing. Note that although we are sorting the source and destination vertices when partitioning graphs, GraphReduce is able to take any user-provided partitioning logic as a plug-in to the *Partition Logic Table* inside the Partition Engine (Figure 31).

3.3.3 Data Movement Engine

To address the cost of data movement caused by accesses to and updates of edge/vertex states, the Data Movement Engine (shown in Figure 32 as **2**) in GraphReduce seeks to accelerate data movement via asynchronous memory-copy operations for concurrent GPU kernel execution. For instance, for NVIDIA GPUs, the programming environment allows

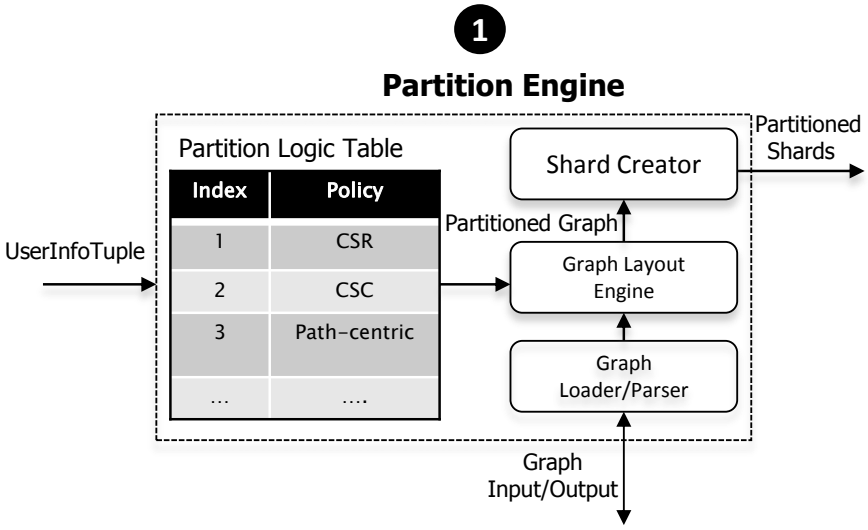

Figure 31: The structure of the Partition Engine.

the concurrent execution of operations from the same GPU protection domain or context. A sequence of operations that execute in issue-order on the GPU is defined as a *Stream Object* [4]. Operations from multiple *Streams* can be executed concurrently and interleaved, leveraging the parallelism provided by multiple hardware queues (e.g., Hyper-Qs [13] provided by NVIDIA Kepler architectures shown in Figure 35(a); they permit host to launch multiple concurrent kernels onto a single GPU). In GraphReduce, multiple intervals (and their associated shards from the Partition Engine) can also be concurrently processed by different *Streams*, to obtain a high degree of parallelism. For different shards, each *Stream* spawned by the *Static Stream Creator* inside the Data Movement Engine in Figure 32 typically issues multiple MemcpyAsync() operations and graph computation kernels asynchronously, overlapping data transfer with computation time. In the Data Movement Engine, *Streams* are scheduled and ordered, with the goal to maximize concurrency in data transfer and computation across different shards of the graph.

We now show how to derive the optimal number of shards being transferred concurrently, to maximize the use of PCIe bandwidth. Assuming that shard size is sufficiently large to saturate PCIe bandwidth (since we are dealing with large graphs), we determine the optimal number of shards transferred concurrently as a function of concurrency (number of concurrent operations). Specifically, we define P as the total number of shards, G as the

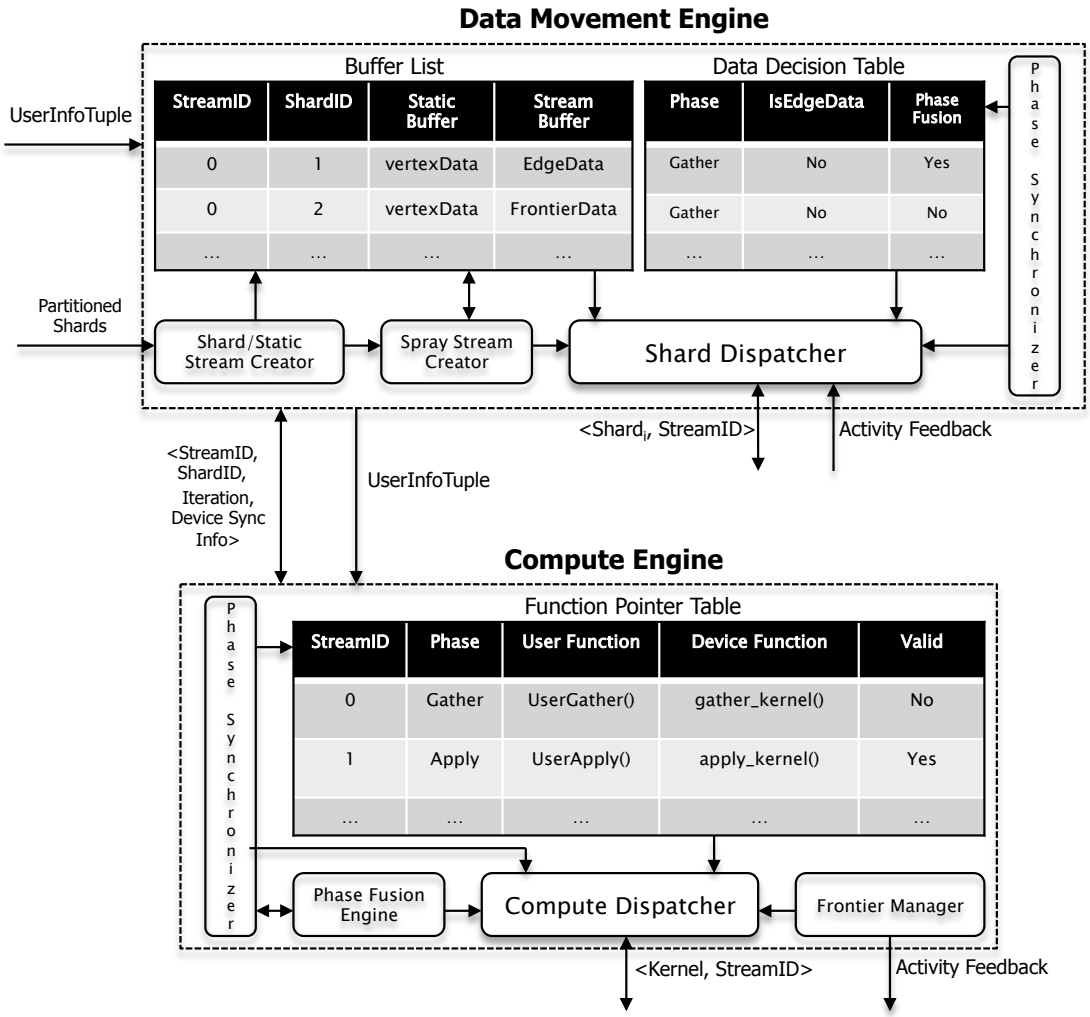

Figure 32: The structures of the Data Movement Engine and Compute Engine. Tables/buffer_list are data structures (passive elements of the engine) while rectangles are modules (active elements of the engine).

size of the entire graph including vertices and edges, V as the size of the vertex set, E as the size of the edge set, K as the total number of concurrent *Streams*; M as the GPU memory size; and B as the bytes needed to achieve maximum PCIe bandwidth. We will have:

$$K * (V/P) + K * B \leq M \tag{5}$$

$$B = (\alpha \times |E| + \beta \times |V|) \tag{6}$$

where the upper bound of K depends on the GPU architecture (e.g., $K \leq 32$ in K20 NVIDIA GPUs), and α and β are the number of edge- and vertex-set streaming buffers (discussed in Section 3.2.2) used in each Stream. In Equation (5), V/P is the size of the interval of the vertex set for one partition. B is the minimum buffer size to saturate PCIe bandwidth (we assume each shard is big enough to saturate that bandwidth). Unknown parameters are K and P, of which P can be derived from fixing the shard size to maximum PCIe bandwidth (Equation (6)). With the limited GPU memory size M, the maximum number of concurrent transfers is bounded by the size of vertex interval plus the sizes of concurrent shards. For instance, based on (5) and (6), we can estimate the optimal number of shards being transferred concurrently to be 2 for our NVIDIA K20 Kepler GPU with 4.8 GB memory.

3.3.4 Computation Engine

The *Computation Engine* shown as **3** in Figure 32 is mainly responsible for GPU in-memory computation (i.e., parallelize each phase of the GAS model) and to send feedback information to the *Data Movement Engine* about the computation frontier used for the next iteration (discussed in Section 3.1.2) .

Recall the shard data structure illustrated in Figure 29, where edge_update_array and vertex_update_array are the update lists of the vertices and their in-edges, respectively, in the corresponding interval (or shard). They store the updates from the *Gather* and *Scatter* phases of the programming model, shown in Figure 33. With these data structures at the very top level, GraphReduce implements a variation of the GAS programming model [80, 56, 77], shown in Figure 34. It includes the following five phases, where every iteration is over all shards instead of all edges:

- *gatherMap*: this function fetches all the updates and messages along the in-edges, getting each edge the state of the source vertex and updating that in the edge_update_array or GatherTemp data structure.
- *gatherReduce*: reduces all the collected updates for each vertex with the reduction function defined by programmers.

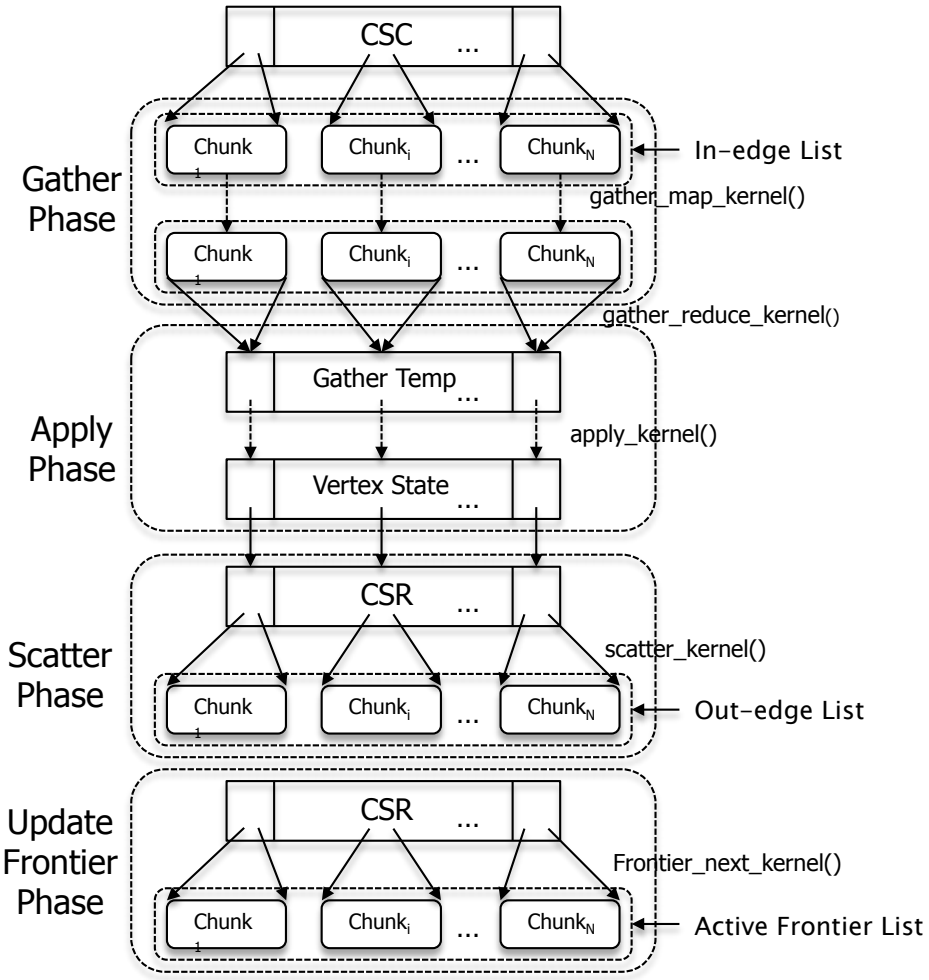

Figure 33: Sub-phases of the computation stage.

- *apply*: applies reduced updates to each vertex to obtain new states for the vertices.
- *scatter*: distributes the updated states of the vertices along the out-edges, i.e., update the edge states of the out-edges of the vertices in each shard. In this phase, only the edge states are updated (if the algorithm allows mutable edge states).
- FrontierActivate: marks the set of edges and vertices that would be active in the next iteration. *This phase is not user-defined but auto-generated by the GR framework.*

For each phase in Figure 34, GraphReduce requires memcpy-in and memcpy-out operations so as to process all shards of the entire graph. The next phase will not start until the previous phase has been completed (following the model of Bulk Synchronous Parallelism).

GPU Device Code for 2-level Parallelism

```
Gather:
Gather_map:
    parfor each in_edge_array e
     update to edge_update_array using gatherMap(vertex_array[e.src])
deviceSynchronizeBarrier()

Gather_reduce:
      parfor each vertex v in the interval:
       update vertex_update_array list using
                  gatherReduce(edge_update_array, UserGatherReduce)
Apply:
         parfor each vertex v in the interval:
         update vertex state of v using apply(vertex_update_array[v.val])
         if(v.val updated)
               v.active = true
Scatter:
         parfor each out_edge_array e in the shard:
         update edge state of e using scatter(e)
FrontierActivate:
         parfor each out_edge_array e in the shard:
         if(e.src.active = true)
               activity_list = activity_list U {e.dst}
```

Figure 34: GPU device pseudo code for exploiting two-level parallelism in different phases.

This can be optimized through dynamic phase fusion and elimination through the *Phase Fusion Engine* inside the Compute Engine, discussed in Section 3.4.3.

Parallelism in different phases exists at two levels. First, operations are run concurrently within each shard in a given phase. Second, in a given phase, computation across different shards can also be executed concurrently (i.e., multiple shards residing in GPU memory at the same time), because there are no data dependencies between shards in the same phase. The device code in Figure 34 shows how GraphReduce exploits this two-level parallelism. This also motivates the use of a hybrid programming model explained in Section 3.2.1, as we use an edge-centric implementation for gatherMap, scatter, and FrontierActivate phases, but use a vertex-centric implementation for the gatherReduce and apply phases.

Note that the *Function Pointer Table* inside the Compute Engine can take user-defined load-balancing strategies as plug-ins. In the current version of GraphReduce, we apply *CTA (Cooperative Thread Array) load balancing* from the Modern GPU library [10]. Scan operations and mergesorts are also implemented using the Modern GPU library.

Figure 32 also shows the data structures and modules embedded in **3.**

(1) **Function Pointer Table**: With adaptivity in mind, Function Pointer Table can take user-defined optimizations as plug-ins. In the current GraphReduce, we apply *CTA load balancing* [10]. Scan operations and mergesorts are implemented using modern GPU library [10].

(2) **Frontier Manager**: It maintains a list of vertices whose states have changed in the current iteration, and then determines the set of vertices which will be active in the next iteration (details in Section 3.4.2).

(3) **Phase Fusion and Elimination**: analyzes if the phases can be merged or even eliminated and based on the phase fusion decision from the Data Movement Engine via phase Synchronization module (details in Section 3.4.3).

(4) **Compute Dispatcher**: It is the core of the Compute Engine which offloads computation operations onto GPU. It is also responsible for using the activity information of the edges to load balance the threads.

(5) **Phase Synchronizer**: It is the control unit manages computation phases and iterations. It is responsible for the synchronization between the compute and data modules, and updating the decision table for fusion/phase elimination optimizations.

3.4 Optimizations

3.4.1 Asynchronous Execution and the Spray Operation

GraphReduce asynchronously performs computation and communication. Specifically, it leverages CUDA Streams, double buffering, and hardware support like Hyper-Qs provided by architectures like NVIDIA's Kepler, to enable data streaming and computation in parallel. As shown in Figure 32, the *Static Stream Creator* of the Data Movement Engine spawns separate CUDA Streams to launch multiple kernels and to transfer data asynchronously,

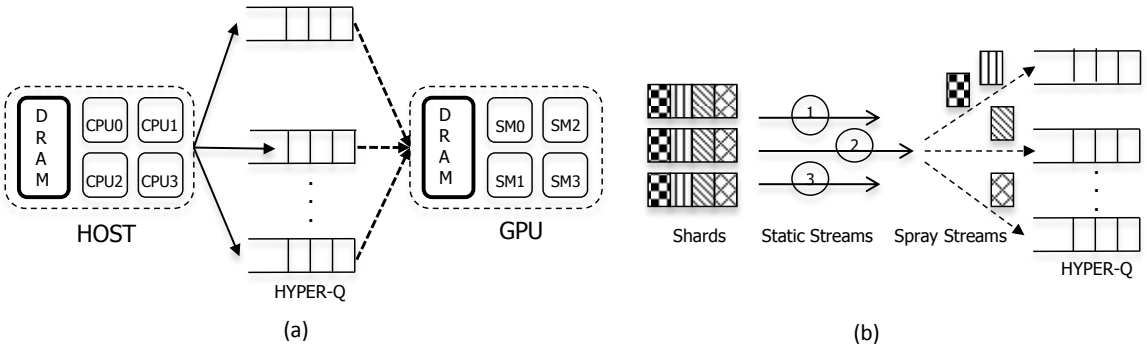

Figure 35: (a) Data Movement from host to GPU in GraphReduce through Hyper-Q. (b) Illustration of Spray Streams for better throughput.

overlapping memory copies within and across phases of computation. Novelty in GraphReduce is its use of separate CUDA Streams for **deep copy operations**, in order to take advantage of the large number of hardware queues offered by modern GPU architectures. This is motivated by the fact that a shard in GraphReduce is not a single contiguous byte-array, but consists of many sub-arrays containing edge, vertex, and frontier data. Each of these sub-arrays requires a separate deep copy to move data between GPU and host. GraphReduce exploits this fact, as shown in Figure 35(b), by not moving the entire shard in one copy performed by a single CUDA stream, but instead, the *Spray Stream Creator* in **2** dynamically spawns multiple CUDA Streams to move these sub-arrays to the GPU. The outcome is concurrent use of the GPU's many hardware queues, which consequently improves overall throughput.

3.4.2 Dynamic Frontier Management

As discussed in Section 3.1.2, with irregularity in graph processing, in every computation iteration, the frontier size is not constant, varying with graph algorithms and datasets. For instance, as shown in Figure 25(c), for dataset Cage-15 processed through BFS, only one vertex is active for the first iteration. Inactive vertices or edges in each iteration can result in significant performance degradation due to GPU thread idling. To address this problem, we integrate the *Frontier Manager module* into the Computation Engine to maintain the list of active vertices whose states have changed in the current iteration and uses it to determine the set of vertices in one hop neighborhood that will be active in the next iteration (based

on out-edges information in a shard). GraphReduce then uses this frontier information of the *next* iteration to avoid unnecessary memcpys and kernel launching. It also uses the active vertex information of the *current* iteration for CTA load balancing to avoid GPU core idling.

3.4.3 Dynamic Phase Fusion/Elimination

A graph algorithm implemented with GraphReduce need not implement user-defined functions for all three GAS phases. For example, BFS need not implement the Scatter phase. In response, when a phase is elided by the user, GraphReduce eliminates the repeated and unnecessary movement of shards into GPU memory, before and after that phase, in each iteration. This is termed phase elimination. For instance, if the graph algorithm does not have a defined gather function, GraphReduce will avoid bringing in the entire shard (in-edges+out-edges), only moving the out-edges to the GPU memory (because in-edges are used only in the Gather phase). Out-edges are moved regardless, because the FrontierActivate phase operates even when there is no scatter phase defined. The resulting dynamic phase elimination reduces unnecessary kernel launching and data movement.

In certain scenarios, merging two or more GAS phases is possible, again to avoid unnecessary extra data movement. For example, if a graph algorithm only defines Apply and FrontierActivate phases, GraphReduce will automatically merge these two phases, thus avoiding the *memcopy* operations that would have been required for executing the two phases separately. We term this action dynamic phase fusion. An example of a graph algorithm for which this method is used is again BFS. It only requires users to define the apply phase, in which the BFS tree depth for every vertex is marked to be the iteration number. GraphReduce will automatically merge the Apply and FrontierActivate phases for BFS. Note that Dynamic Fusion/Elimination functionalities are enabled through the *Phase Fusion Engine* inside the Compute Engine.

Other minor optimizations introduced in the GR frame- work include: 1) if the shards fit in the memory then there is no need to write back the edge state to the host. Subsequent phases and iterations reuse the shard in its computation. Similarly, if the vertex array

fits in the memory then no need to write it back to the host after each iteration. 2) avoid unnecessary write back of temporary states like the gatherTemp to host which will be overwritten in the next iteration anyway. To implement this GR uses the read/write attribute information from the Buffer-list.

3.5 Experimental Evaluation

3.5.1 Experimental Setup

Evaluation Platform: GraphReduce is evaluated on a typical heterogeneous HPC node equipped with 16-core Intel Xeon E5-2670 processors running at 2.6 GHz with 32 GB of DDR3 RAM, and one attached NVIDIA Tesla K20c GPU with 13 SMX multiprocessors and 4.8 GB GDDR5 RAM. The Kepler GPU is enabled with CUDA 6.5 runtime and the version 340.29 driver, while the host CPU side is running Fedora version 20 with kernel v.3.11.10-301 x86. All the runs are compiled with the highest optimization level flag.

Graph Dataset. Shown in Table 3, we evaluate the performance and efficiency of GraphReduce using two types of graph inputs: small size graphs that will fit into GPU memory (named In-memory graphs) and large graphs that do not fit (named Out-of-core graphs). Here, we define the size of a graph as the amount of memory required to store the edges, vertices, and edge/vertex data states in terms of the user-defined datatypes and a few of the temporary buffers. All experiments use datatype *float*. Note that the size of a graph can expand after loading it to in-memory buffers, because the size of the datatypes for edge and vertex states is in general larger than their representations in the raw graph format (e.g., *char*).

In-memory graphs are used to evaluate the effectiveness of GraphReduce's in-memory optimizations against other state-of-the-art in-memory approaches (e.g., MapGraph and CuSha), while Out-of-core graphs are used to evaluate it against frameworks that can process large graph sets (e.g., GraphChi and X-Stream).

The ten real-world graphs listed in Table 3 are publicly available and cover a wide range of sparsity and sizes. For example, *orkut* is an undirected social network, in which vertices and edges represent the friendship between users. *uk-2002* is a large crawl of the

.uk domains, in which vertices are the pages and edges are the links. *nlpkkt160* is from the 3D PDE-constrained optimization problem with vertices as state variables and edges as control variables.

Evaluated Algorithms. Four widely used algorithms are evaluated, including Breadth First search (BFS), Page Rank (PR), Single-Source Shortest Paths (SSSP), and Connected Components (CC). Algorithms requiring undirected graphs as inputs, e.g., connected components, are stored as pairs of directed edges.

3.5.2 Evaluation and Analysis

Graph		BFS	SSSP	Pagerank	CC
kron-logn21	GraphChi	365	442	328	236
	Xstream	95	97	98	97
	GR	4	7	93	9
nlpkkt160	GraphChi	503	510	447	1560
	Xstream	128	136	144	133
	GR	60	92	140	183
uk-2002	GraphChi	1100	1283	1091	1073
	Xstream	330	374	335	348
	GR	49	80	153	162
orkut	GraphChi	311	320	285	268
	Xstream	124	131	127	127
	GR	6	10	84	16
cage15	GraphChi	262	265	240	389
	Xstream	114	119	115	143
	GR	18	25	19	41

Table 5: Execution times of out-of-core graph processing frameworks on different algorithms and graph inputs. Reported times are wall time and in seconds.

3.5.2.1 Comparison with Out-of-Core Frameworks

Since the state-of-the-art GPU-based graph processing approaches [52, 23, 119, 67] assume that input graphs fit in GPU memory, we compare GR's out-of-core performance with Graphchi [71] and X-Stream [93], both state-of-the-art, out-of-core, CPU-based frameworks targeting large real-world graphs. For fairness in comparison, the datasets chosen (in-memory sizes shown in Table 3) fit in host memory but do not fit into GPU memory. This is to avoid I/O (SSD access) overheads in systems like GraphChi and X-Stream. GR, however, incurs the unavoidable costs of moving *shards* in and out of GPU memory.

Graph		BFS	SSSP	Pagerank	CC
ak2010	MG	7.94	79.01	23.86	19.03
	CuSha	7.75	31.99	12.08	10.16
	GR	9.26	3.81	14.61	17.78
coAuthorsDBLP	MG	5.28	8.75	68.92	30.26
	CuSha	11.55	12.75	79.84	13.99
	GR	5.31	5.42	53.14	16.43
kron-logn20	MG	51.81	139.43	6789	308.91
	CuSha	119.82	269.88	1852	138.7
	GR	27.88	28.34	4365	266.86
webbase-1M	MG	8.71	13.56	72.86	50.97
	CuSha	13.52	12.65	270.83	317.41
	GR	1.4	6.07	57.76	37.45
belgium_osm	MG	195.79	261.32	102.64	2219
	CuSha	791.3	897.03	45.8	920.7
	GR	279.8	281.39	71.33	40.63

Table 6: Performance results of in-memory (small) graph processing frameworks on different algorithms and graph inputs. Reported times are in milliseconds. MG stands for MapGraph.

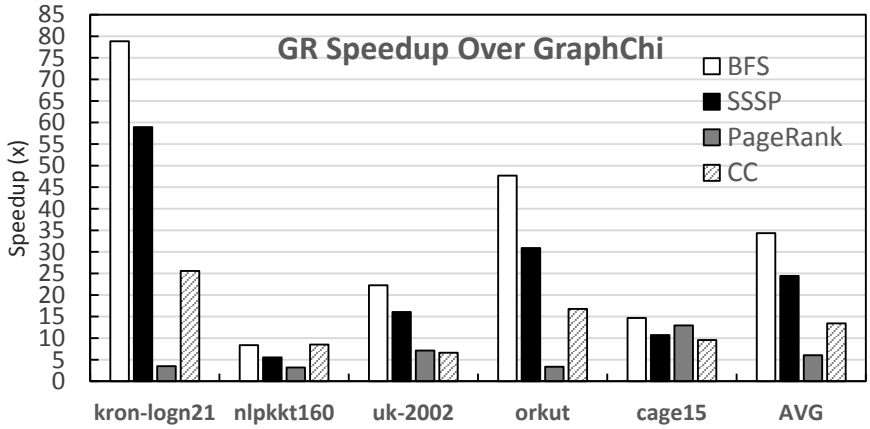

Figure 36: GR's speedup over GraphChi for various algorithms and out-of-core graph inputs.

Shown in Table 5, Figure 36, and Figure 37, GR achieves an average speedup of **13.4x and 5x** over GraphChi and X-Stream (running with 16 threads), respectively, despite its need to move data between GPU and CPU via PCIe; while Graphchi and X-Stream benefit from local (host) memory access. GR achieves some significant speedups, e.g., up to 79x over GraphChi and 21x over X-Stream, for kron_g500-logn21 processed by BFS. These performance improvements are due to its (i) asynchronous mode and spray operation (leveraging CUDA Streams, Hyper-Qs, and deep memory copy operations); (ii) dynamic frontier

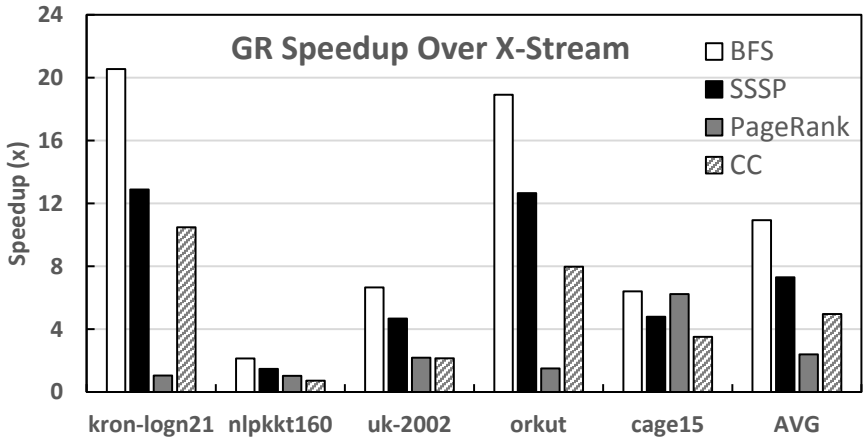

Figure 37: GR's speedup over X-Stream for various algorithms and out-of-core graph inputs.

management, to avoid unnecessary kernel launching and GPU core idling; and (iii) dynamic phase fusion/elimination to remove unnecessary data movement. The hybrid programming model also contributes to the performance improvements over GraphChi and X-Stream, by extracting access pattern-based parallelism opportunities across different phases. GraphChi (vertex-centric) and X-Stream (edge-centric), on the other hand, suffer from significant random accesses to either their edge or vertex sets, due to their use of a unified model. There is only one case that X-Stream performs slightly better than GR, which is the the nlpkkt160 input processed by CC. This is due to the fact that GR experiences substantial overheads from the large data movement over PCIe, and these overheads are not sufficiently compensated by the massive parallelism offered by GPU. GrapChi and X-Stream, in comparison, have all data accessible locally in the host memory and are therefore, not subject to such overheads.

3.5.2.2 Comparison with GPU In-Memory Frameworks

The results above establish GR's ability to process large graphs that do not fit into GPU memory, at levels of performance higher than that seen for CPU-based solutions. In other words, additional costs arising from GPU-host data movement are typically dwarfed by the performance advantages offered by fast GPUs. At the same time, GR also performs as well as the existing in-GPU-memory solutions for smaller graph inputs. Table 6 shows

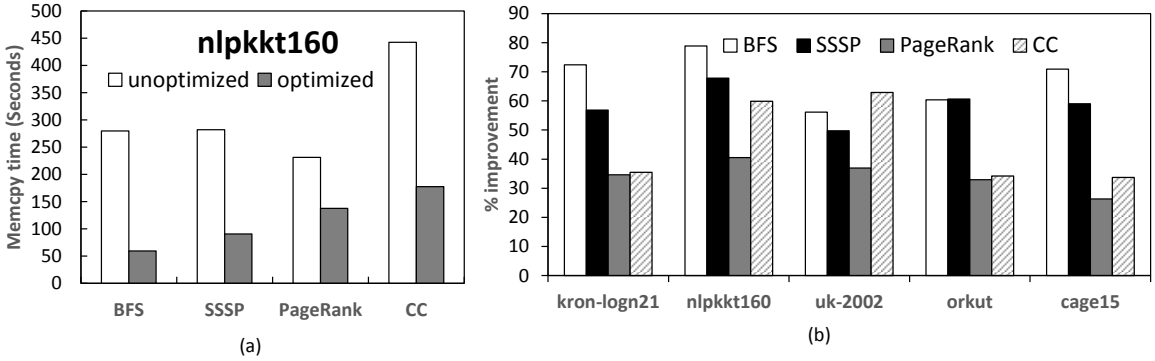

Figure 38: Performance gained from memcpy optimization. (a) Actual memcpy time comparison between optimized and unoptimized GR for nlpktt160. (b) Percentage improvement of memcpy performance from optimized GR against unoptimized GR.

GR's in-memory performance for smaller graphs to be comparable to the state-of-the-art in-memory processing frameworks like MapGraph (MG) and CuSha, which apply multi-level fine-tuned optimizations for in-GPU workloads. In many cases, GR outperforms MG and CuSha significantly, e.g., kron_g500-logn20 with SSSP and webbase-1M with BFS. For processing these smaller graphs, one major contributing factor for high performance in GR is its use of active vertex information of the same iteration for the CTA load balancing. One interesting observation is that not all of the GPU in-memory graph processing approaches work well for every graph input and algorithm. This prompts us to (also see Sections 3.3.2 and 3.3.4) to add flexibility to GR's Partition Engine – the Partition Logic Table can be easily modified to incorporate desired user-defined specific optimizations, e.g., to use the partition and graph layout algorithms employed in CuSha.

3.5.2.3 Performance Effects of GraphReduce Optimizations

Experimental results show memcpy time to be a dominant factor for performance, occupying on the average above 95% of the total execution time for the five large out-of-core graph inputs studied above. This makes it the primary target for GR optimizations. Figure 38 shows the performance improvements gained from the three optimizations discussed in Section 3.4, including asynchronous execution/spray operation, dynamic frontier management, and dynamic phase fusion/elimination. For example, without these optimizations, Figure 38(a) shows that the performance of nlpkkt160 suffers significantly from memcpy, e.g., up

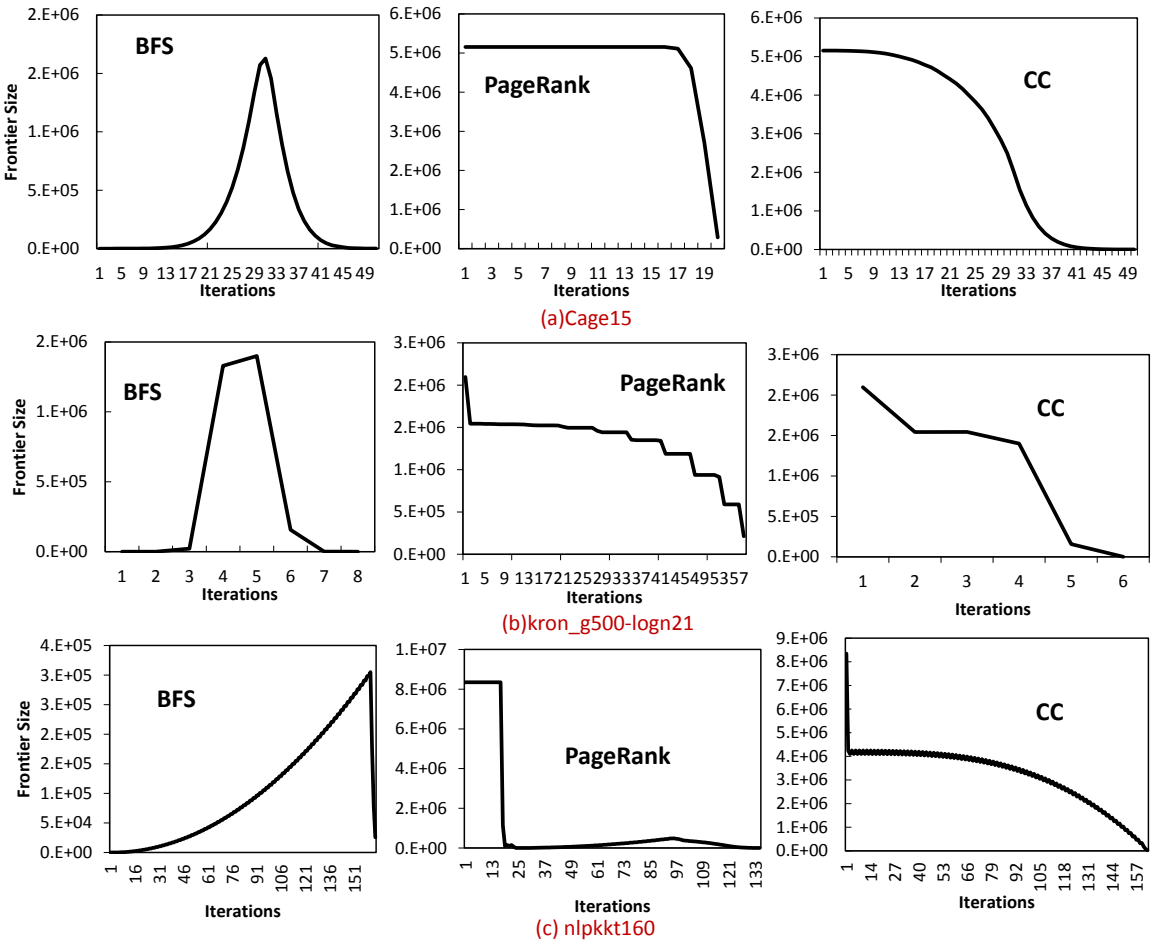

Figure 39: Frontier size changes across iterations shown for several large out-of-core graphs with three algorithms.

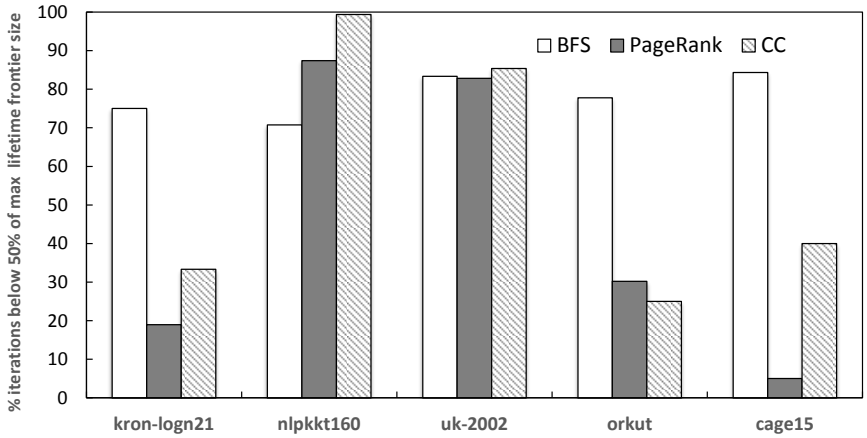

Figure 40: For out-of-core graphs, percentage of iterations that are below 50% of the max lifetime frontier size.

to 443 seconds for the CC with the total execution time only being 451 seconds. With the optimizations, memcpy time drops from 443 seconds to 178 seconds, improved by 60%. Figure 38(b) shows the percentage improvement of memcpy time from the optimized GR over the baseline unoptimized scenario, with an average of 51.5% and up to 78.8 % across all large datasets and algorithms.

A simple example shows how dynamic frontier management affects memcpy time. Figure 39 shows that frontier sizes vary with the iterations for three large graph inputs and three algorithms (Note: SSSP is not included here because the frontier patterns for BFS and SSSP are very similar as BFS is essentially SSSP with equal edge weights). This indicates that the basic pattern (or shape) of the frontier graphs is algorithm-dependent, while the rate at which the frontier size changes with iterations is input-dependent. Figure 40 shows the percentage of iterations with active vertices that are below 50% of the max lifetime frontier size (the peak value shown in Figure 39) across five large data graphs and three algorithms. Combining this figure with Figure 38(b), we can observe that graph inputs with higher percentage of iterations with small frontier sizes benefit the most from the dynamic frontier management in GR, e.g., cage15 with BFS and uk-2002 with CC.

3.5.2.4 Discussion

Experiments demonstrate that (1) GraphReduce can process graphs of sizes larger than GPU memory, achieving up to 79x and 21x, and an average of 13.4x and 5x, speedup over the competing CPU-based methods implemented in GraphChi and X-Stream respectively, for several real-world large-scale graphs with various algorithms. (2) GR performance is comparable to that of existing in-GPU- memory solutions like MapGraph and CuSha, for smaller input graphs (i.e., those that fit into GPU memory). (3) Memcpy time is the dominant factor in GR's graph processing, occupying on the average above 95% of total execution time. Because there is a strong correlation between the change in active vertices per iteration vs. the amount of unnecessary data movement, the more inactive vertices there are per iteration, the more opportunities exist to avoid such unnecessary data copies. This is evident from performance results showing the effects of GR's dynamic frontier management.

Finally, (4) GR employs additional optimizations that include concurrent copy operations, overlapping computation and communication operations; deep copies via spray streams, leveraging the multiple hardware queues (Hyper-Qs) in GPUs; and performs dynamic phase merging and elimination to avoid unnecessary data copying. With these optimizations, GR achieves an average of 51.5% and up to 78.8% reduction in memcpy time across the large datasets and algorithms used in our evaluation.

3.6 Chapter Summary

In contrast to the previous work, GraphReduce (GAS model based) is able to process graphs of sizes much exceeding that of GPU memory, by sharding graph data and asynchronously moving shards between GPU and host memories. Technical advances offered by GraphReduce include its usage of a hybrid programming model of edge- and vertex-centric processing, asynchronous execution/spray operation, dynamic phase fusion/elimination, and dynamic frontier management. With these optimizations, GraphReduce achieves levels of performance similar to those of prior in-GPU-memory and significant speedup over out-of-core implementations of graph processing like GraphChi and X-Stream respectively, for several real-world large-scale graphs processed by various algorithms. Further, as a framework, GraphReduce permits the usage of alternative data partitioning schemes and associated data layout methods, thereby enabling extensions that can take advantage of the state-of-the-art schemes for graph processing developed elsewhere.

There are several interesting future directions of our work, including: (1) extending GraphReduce to support multiple on-node GPUs, (2) addressing the limited on-node memory size through the usage of SSD and other storage devices; (3) processing dynamically evolving graphs; (4) understanding how dynamic profiling and processor choice (i.e., GPU vs. CPU execution) [51] could be integrated into GraphReduce; and (5) adapting architectural- and runtime-level optimizations to further improve performance and energy efficiency of the highly irregular graph algorithm [75, 108, 107].

CHAPTER IV

EVOGRAPH: PROCESSING EVOLVING GRAPHS ON ACCELERATOR-BASED SYSTEMS

As discussed in last chapter, a recent trend is the gain in popularity of GPU processing in many domains such as social networks, e-commerce, advertising, and genomics. This has motivated the growing interest in large-scale real-world graph processing for both scientific and commercial applications, as well as the recent efforts in accelerator-based graph processing frameworks such as MapGraph [52], Medusa [119], CuSha [67], GraphReduce [98], and so on. An important aspect of real-world graphs, like Facebook friend lists or Twitter follower graphs, is that they are dynamic. Given the billions of Facebook [28] users sharing more than 100 billion photos and posts per month, let alone the volume on Twitter [19] or other blogging platforms, there is a huge need to quickly analyze this high velocity stream of graph data.

However, state-of-the-art graph analytics for dynamic graphs follow a store-and-static-compute model that involves batching these updates into discrete time intervals, applying all of the updates to the total graph, and then rerunning the static analysis. There is considerable redundancy and inefficiency in this approach to analyzing this *evolving* graph sequence. Static graph analytics on a single version of the evolving graph, even when leveraging massive amount of parallelism offered by thousands of cores in a GPU, can be very slow due to the extreme scale of many real-world graphs (e.g., one Facebook graph purportedly has a trillion edges [40]) and/or because of the complexity of the graph queries that are traditionally both compute and memory intensive. Second, data movement of the entire input graph repeatedly between the host and the GPU over the slow PCIe link can result in substantial overhead, which in turn can overshadow the benefits from the massive parallelism offered by a GPU. Finally, there are real world graph analytics problems that inherently require soft or hard real-time guarantees, e.g., real-time anomaly detection,

disease spreading, etc, and hence have difficulty using the traditional static recomputation model. Beyond just hardware performance, we also note that the skills to write performant GPU code are substantially different from the coding skills that many analysts have learned. As one can therefore see, the many demands of high velocity graph data, both commercial and scientific, have outstripped the traditional, batched static graph analytics models, even when using GPUs.

To address this, we propose a two-pronged approach to deal with both the performance and programmability challenges. We introduce an accelerator-based incremental graph processing framework called EvoGraph. EvoGraph employs a new variant of the popular Gather-Apply-Scatter (GAS) programming model, which we call Incremental-GAS (or I-GAS), to incrementally process a batched stream of updates (i.e., edge/vertex insertions and deletions). The key insight is that I-GAS algorithms are designed to work over a (dynamically determined) sub-graph of the previous version of the evolving graph. For many popular graph algorithms and real-world graphs, the corresponding I-GAS logic affects only a fractional portion of the graph; this reduction in problem size can result in large performance benefits compared to static recomputation of the graph algorithm on the entire graph. The modest additions of the I-GAS model to the already-published GAS model interface enable an easy transition of analysts from coding in a static to a dynamic streaming environment.

From a simplistic view, it would seem that incremental methods would always be preferable. However, there are scenarios when a streamed update may affect a very large portion of the graph, and incremental processing won't help much or may even be worse than static recomputation due to overheads of incremental execution. One such counterexample is in the incremental version of Breadth First Search (BFS), where updates that affect vertices close to the source/root node will affect nearly the entire BFS tree. Here the incremental run can at best perform as good as the static re-run, and may even be worse. Note that this is not a concern about correctness of the result, but over performance. Therefore, in order to handle such scenarios, we employ a *per batch*, property-based dynamic choice between incremental and static graph processing called *property-guard*. Utilizing user-defined

and built-in properties (e.g., vertex degree) along with programmable control policies, EvoGraph analyzes each update batch and dynamically decides whether to process the graph incrementally using I-GAS or to fall back to static recomputation.

This chapter makes the following technical contributions:

- *EvoGraph*, an accelerator-based high-performance incremental graph processing framework, built on top of GraphReduce [95], to process evolving graphs by avoiding the naive static graph recomputation approach. User's sequential code for incremental graph algorithms in EvoGraph are seamlessly mapped to GPU for acceleration.

- Improved GPU core utilization via dynamic merging of GPU contexts from different graph applications, and additional hardware parallelism extracted using deep copy operations on separate CUDA streams to leverage the multiple hardware queues (Hyper-Qs) in GPUs.

- An extensive evaluation of 3 popular graph algorithms on real-world and synthetic graph datasets show that EvoGraph can significantly outperform the existing static recomputation approach. Compared to competitive frameworks like STINGER, EvoGraph achieves a performance improvement of up to 232x and overall throughput of 429 million updates/sec.

- Graph-property-based performance optimization called *property-guard* to dynamically decide between static and dynamic graph execution based on user-defined and built-in graph properties, resulting in a speedup of up to 18.4x over a naive streaming approach.

The remainder of the chapter is organized as follows: Section 4.1 discusses the motivation and challenges of evolving graph analytics. Section 4.2 dissects the design choices. Section 4.3 introduces our EvoGraph runtime framework. Section 4.4 highlights case study of three classes of incremental graph algorithms implemented using EvoGraph. Section 4.5 presents the experimental setup and result analysis followed by conclusions and future work.

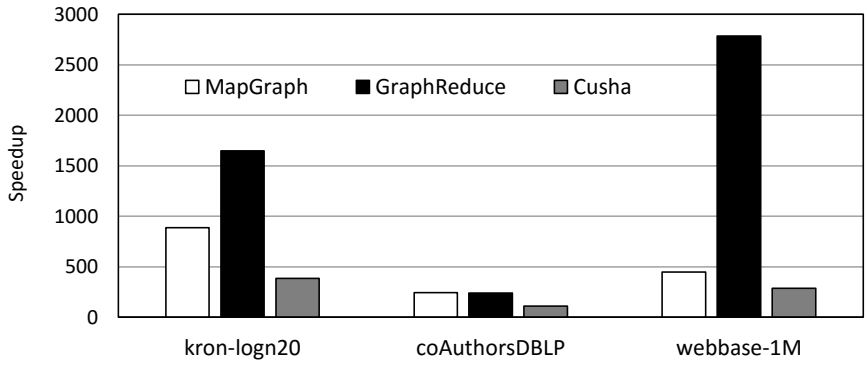

Figure 41: State-of-the-art GPU frameworks (i.e., MapGraph, GraphReduce and Cusha) for processing static graphs significantly outperform the best CPU-based framework X-Stream (baseline).

4.1 Motivation and Challenges

GPUs have emerged as one of the most powerful computation accelerators for world-class supercomputers [18] because of their unparalleled massive amount of parallelism and ability to speedup a wide range of HPC applications. Compared to its counterpart CPU, it also often provides superior acceleration for general graph algorithms. Figure 41 demonstrates that for processing three real-wold in-memory *static graphs* under BFS, state-of-the-art GPU frameworks outperform the best CPU-based graph analytics (i.e., X-Stream) by a speedup of up to 2785x (i.e., GraphReduce on *webbase-1M*). This motivates us to unleash GPU's high computation power for processing *evolving dynamic graphs*.

Figure 42 shows an example of an evolving Linkedin social network graph, in which a subgraph (circled by red dashed line) is going through *update batches* (e.g., insert:(1,4) and delete:(1,3)) at different time point. Different colors of dots represent work fields. Processing such common constantly-evolving social network graphs on GPUs is very challenging because (i) highly efficient computation model and convenient programming constructs do not exist for programmers to effectively express their algorithms on GPUs, (ii) how to efficiently utilize the parallelism provided by GPUs to deal with the computation and data storage overlap in dynamic graphs is complicated, and (iii) how to extract the most throughput from GPUs without burdening the users with hardware details is unclear. In order to address these challenges, we designed a runtime graph analytics framework named *EvoGraph*

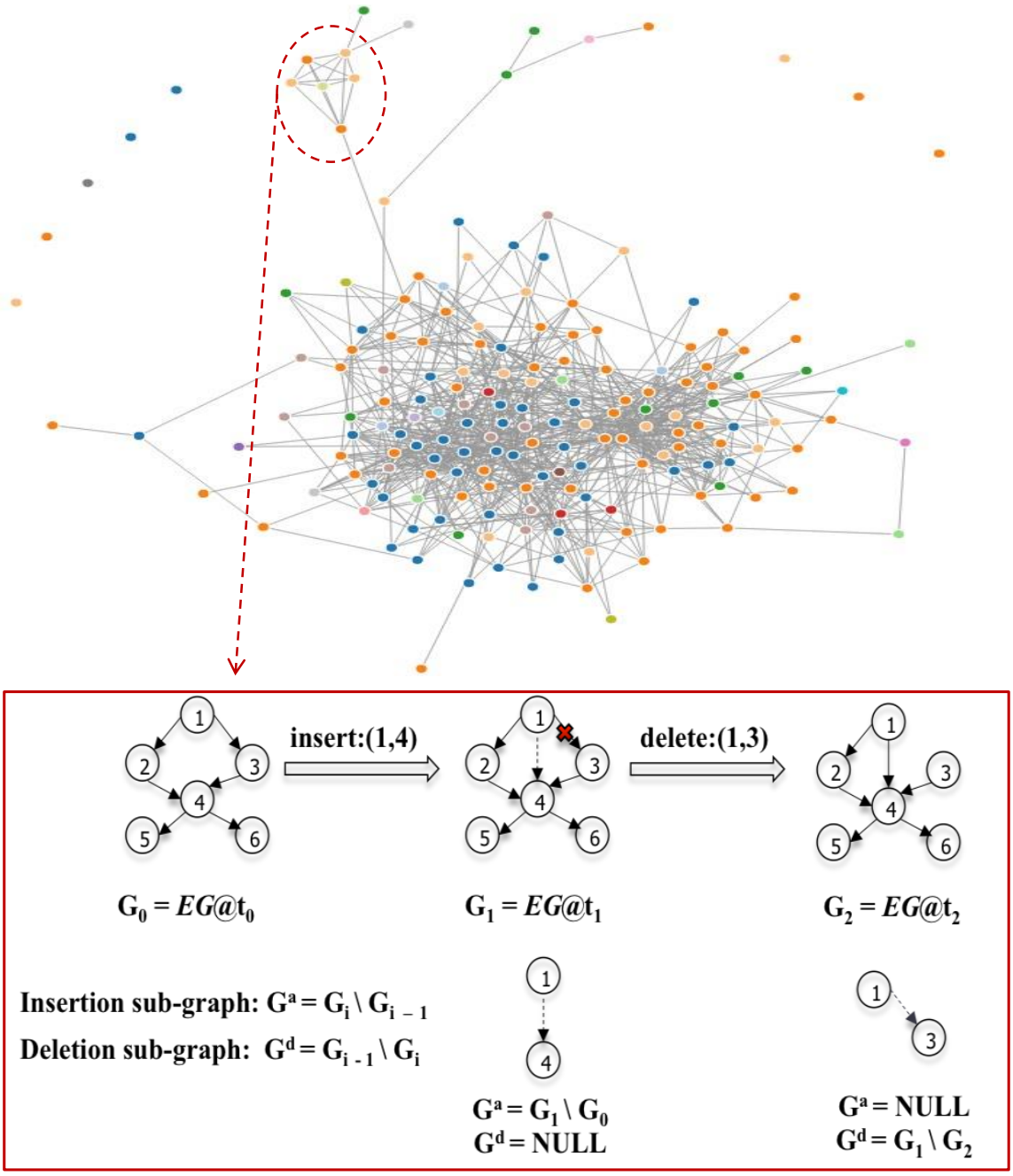

Figure 42: A subgraph of a Linkedin social network has been updated over time but the rest of the network remains the same.

to process complex evolving graphs on modern GPUs. Under EvoGraph, users only need to write sequential codes and the sophisticated runtime will seamlessly map the incremental graphs to GPU for acceleration. We will discuss the design details of EvoGraph next.

4.2 Design Choices

In general, there are two major strategies for processing evolving graphs: (1) Offline evolving graph processing where multiple versions of the graph are stored and analyzed for the change in certain graph properties over time. (2) Online evolving graph processing that involve real-time continuous query processing over streaming updates on the evolving graph. EvoGraph is a framework designed for the latter.

Broadly, there are three key characteristics of evolving graphs that dictate the design decisions for EvoGraph:

- Computation overlap in a sequence of graph versions
- Data or structural overlap in a sequence of graph versions
- Choice between static and dynamic execution runtime

4.2.1 Computation Overlap and Programming Model

Shown in Figure 42, across multiple versions or snapshots of an evolving graph, the vertex states or values for many vertices remain the same over time, and thus their recomputation is essentially redundant. We define an *inconsistent vertex* as a vertex for which one or more properties are affected when an update batch is applied. For instance, when calculating out-degree of vertices, an insertion or deletion of edge (v_i,v_j) only makes vertex v_i inconsistent. However, under BFS (Breadth-First Search) algorithm, insertion of edge (v_i,v_j) makes v_j and all the vertices that are descendants of v_j *inconsistent*. One can consider the entire vertex set V to be inconsistent by default. But for many real-world evolving graphs (e.g., Linkedin or Facebook social network), changes affect only a very small subset of the graph. Therefore, computing the vertex states only for those inconsistent vertices while maintaining the vertex states for the rest will significantly reduce the computation time. Because of this, we propose **I-GAS** programming model based on the classic GAS abstraction (Section 4.1) for incremental graph processing, which will be discussed in detail in Section 4.3. To reduce overheads, I-GAS builds a group of inconsistent vertex sets and sub-graphs that are affected

by an update batch and then reduce the incremental graph problem to a sub-problem under GAS.

4.2.2 Structural Overlap and Data Structure Choice

Another key observation to make here is that there can be a huge overlap in the edge and vertex sets between consecutive versions of an evolving graph. For instance, if a graph evolved from G to G' during a certain time epoch t and let $\delta_1 = G' - G$ (insertions), $\delta_2 = G - G'$ (deletions) then $G \cap G' = G - \delta_2 = G' - \delta_1$ is the structural overlap between the two consecutive versions.

Furthermore, there are multiple options for choosing data structure to store an evolving graph. Assume the graph has n vertices and m edges at certain time point. *Adjacency matrices* allow fast update (i.e., $O(1)$ time cost) with both insertions and deletions but require $O(n^2)$ space. *Adjacency lists* are space efficient ($O(m+n)$) and allow fast update, but graph traversals are very inefficient due to non-contiguous memory nodes in the adjacency edge list. *Compressed Sparse Row* (CSR) [33] formats provide both space efficiency and fast traversal through storing offsets rather than all the valid fields in the adjacency matrix. But its insertion and deletion are very expensive because each update requires shifting of the graph data throughout the compressed array to match the compressed format. In order to allow faster updates and process both the incremental and static graph algorithms efficiently, EvoGraph uses a hybrid data structure: edge-lists to store incremental updates and compressed format to store the previous static version of the graph. As mentioned above, the edge-list will allow faster updates without adversely affecting the performance of incremental computation. Meanwhile, the compressed matrix format allows faster parallel computation over the static version of the graph. EvoGraph merges both whenever required (see Section 4.3 for details).

4.2.3 Static vs. Dynamic Runtime

Runtime of online graph analytics varies widely depending on the algorithm and the update. On one hand, there are cases in which incremental algorithms affect only a small local portion of the entire graph (e.g., making a small subset of the graph inconsistent). As

demonstrated in [46], [81], [48], per-vertex properties that depend on a fixed radius affect only a local portion of the graph and hence the runtime is proportional to the update batch size (e.g. triangle counting). On the other hand, there are classes of incremental algorithms whose properties depend on the graph path which may cause a large portion of the graph to be inconsistent, resulting in a complete recomputation of the graph. Under this scenario, incremental processing will not achieve any performance benefit over static recomputation and might even suffer from a performance degradation due to the overheads associated with the incremental execution. To effectively handle both scenarios, EvoGraph applies a heuristic to select the execution pathway: incremental or static. The decision is made dynamically based on a set of built-in or user-defined graph property checks (e.g., vertex degree information) and the fraction of inconsistent vertices in the update batch that meet the criteria. More specifically, if the update is predicted to affect a small portion of the graph then the incremental execution path is taken. Otherwise, the update is merged with the static graph, which will be then recomputed. Take BFS for example. If 90% of the inconsistent vertices in an update batch are of high degree, a large portion of the graph is likely to be impacted, so the static execution path will be taken. The metadata that is used to make decisions on execution path will be discussed in Section 4.3.

4.2.4 Context Merging and Multi-Level GPU Sharing

As mentioned previously, some incremental graph computation only affects a small portion of the graph and hence the GPU cores can be significantly underutilized. This gives us opportunities for GPU resource sharing among static and incremental graph computation. For NVIDIA GPUs, since the CUDA runtime does not allow more than one host processes to share the same GPU context (protection domain), the GPU workloads of two different applications cannot run concurrently on a single GPU. When processing incremental graphs, this could result in high context switching overhead and potential core idling. To avoid this, EvoGraph packs different application contexts into a single protection domain [96], [97] (we call it 'context merging'). Specifically, all the graph applications (static and incremental) collocated on a GPU are mapped to separate host threads of the same per GPU host process,

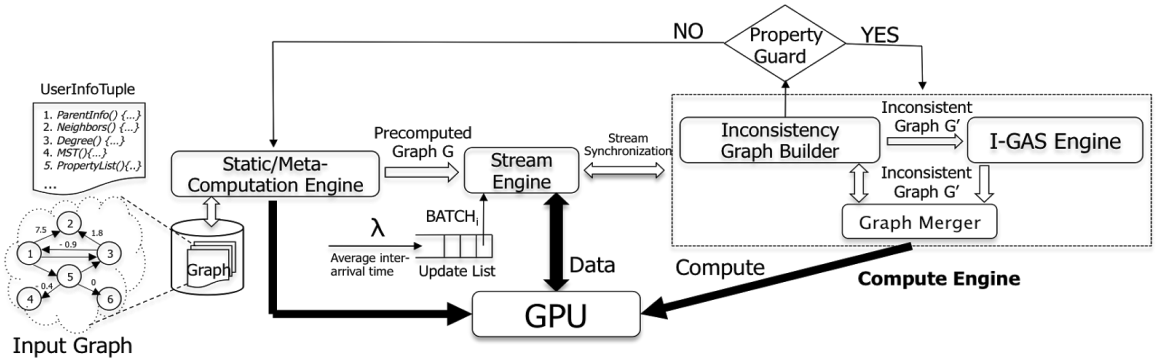

Figure 43: The software architecture diagram of EvoGraph.

with their respective GPU operations invoked via separate CUDA streams. Using this single GPU context to host all applications enables the cross-application sharing of GPU resources - a true multi-tenancy. Additionally, a specific advantage of leveraging CUDA streams is that all three GPU engines, (a) memory copy from host to device (H2D), (b) from device to host (D2H), and (c) computation, can be concurrently executed by different applications. Moreover, graph operations from different applications can also run concurrently on GPU, thereby achieving the benefits of space and time sharing.

4.3 EvoGraph: The Runtime Framework

The EvoGraph framework can efficiently process evolving graphs that incrementally change over time due to edge or vertex insertions and/or deletions by seamlessly mapping the graph computation to leverage thousands of cores available on GPU. The continuous stream of updates is divided into fixed size batches before being processed by EvoGraph in the order of their arrival. EvoGraph simplifies evolving graph analytics programming by supporting a multi-phase, asynchronous, property guarded execution model described in detail in the following sections. Figure 43 shows the general software architecture of EvoGraph which consists of six major components: Static-/Meta-computation Engine, Stream Engine, Inconsistency Graph Builder, I-GAS Engine and Graph Merger. All of these components support GAS-based APIs.

4.3.1 User Interface

Table 7 shows the six user-defined functions for representing the different computation phases in EvoGraph. By customizing these functions, programmers can simply write sequential graph algorithms on the host CPU side. The runtime of EvoGraph will then generate parallelized code to incrementally process evolving graph updates and execute them on the targeted GPU. The user-defined functions include *meta_computation()*, *build_inconsistency_list()*, *property_guard()*, *frontier_activate()*, *update_inconsistency_list()* and *merge_state()*, corresponding to the five computation phases of EvoGraph which are summarized as follows:

1. **Static Graph and Metadata Preprocessing**: computing the static version of the graph and any optional metadata that will be used later for incremental processing.

2. **Marking Out Graph Inconsistency**: creating a list of inconsistent vertices, and optionally, an inconsistent subgraph G'.

3. **Determine the Execution Path by Property Checking**: using the user-defined and built-in property list to examine the current update batch to proactively decide between incremental processing vs. static recomputation.

4. **Incremental GAS (I-GAS)**: applying incremental version of the GAS programming model to move the computational frontier one step per iteration.

5. **State-Merging**: Merging the incremental and static graph states.

As an example, Figure 44 and 45 illustrate the five computation phases in incremental Breath-First Search (BFS) and Connected Component (CC) algorithms of which only CC requiring a separate inconsistent subgraph G' in Phase IV. Before discussing each of these phases in detail, we first look into the **Stream Engine** in Figure 43, which is in charge of optimizing data movement (between host and device) and context merging (discussed in Section 4.2.4).

Table 7: Implementing Graph Algorithms in EvoGraph

Application	GraphIn Phases and APIs						
		Phase I	Phase II	Phase III	Phase IV		Phase V
	Type	*meta_computation()*	*build_inconsistency_list()*	*CheckProperty()*	*frontier_activate()*	*update_inconsistency_list()*	*merge_state()*
Breadth First Search (BFS)	All-merge	Parent id and vertex degree	**1.** Inconsistency list contains vertices with incorrect depth values with MIN_PRIORITY. **2.** G' = G	Check BFS depth property	Activate inconsistent vertices with minimum depth value-Ramalingam and Reps	Remove frontier vertices and add inconsistent successors to inconsistency list	**1.** Apply all insertions and deletions to G.
Connected Components (CC)	Delete-only-merge	Vertex degree	**1.** For each edge insertion add an edge in G' if the endpoints belong to different components. **2.** G' is also known as component graph.	Check disjoint component property	Activate all the vertices in G'	Clear inconsistency list	**1.** Apply only deletions to G. **2.** Relabel components in G using G'
Triangle Counting (TC)	No-merge	Vertex degree	**1.** Inconsistency list contains endpoints of every edge inserted and/or deleted and their respective neighbors. **2.** G' consists of inconsistent vertices and edge incident on them in G.	Check vertex degree property	Activate all the vertices in G'	Clear inconsistency list	**1.** Applying insertions and deletions to G not required **2.** Update triangle counts and degree information in G using G'

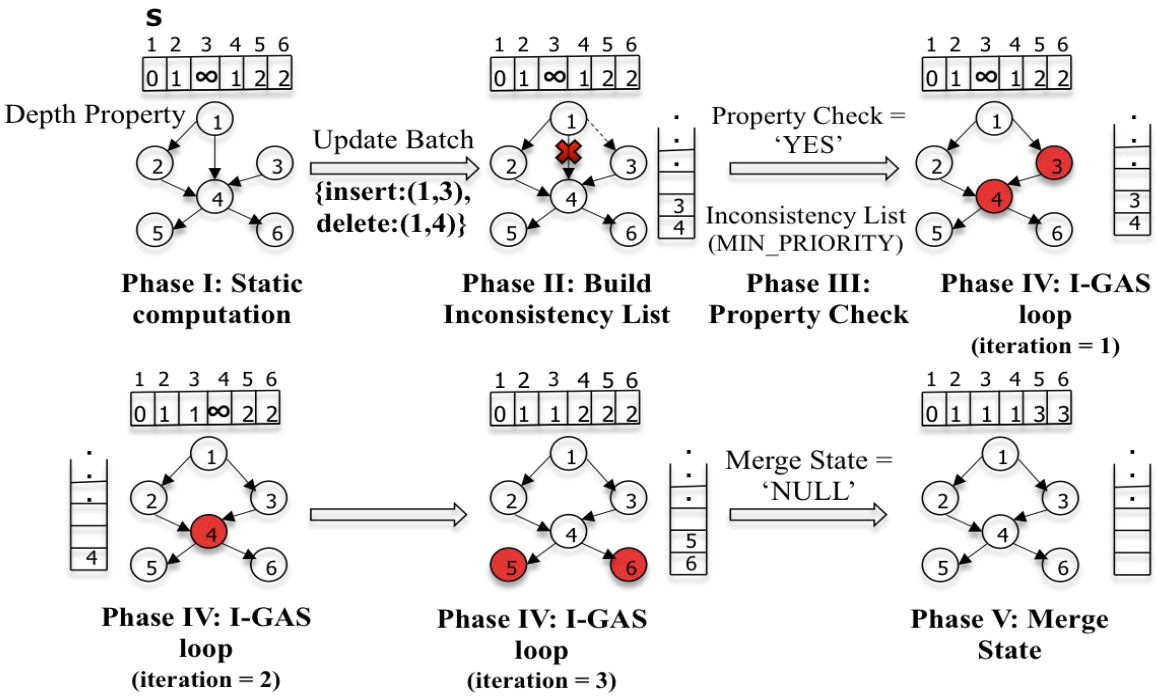

Figure 44: Computation phases of incremental BFS implemented in EvoGraph with inconsistent vertices marked red.

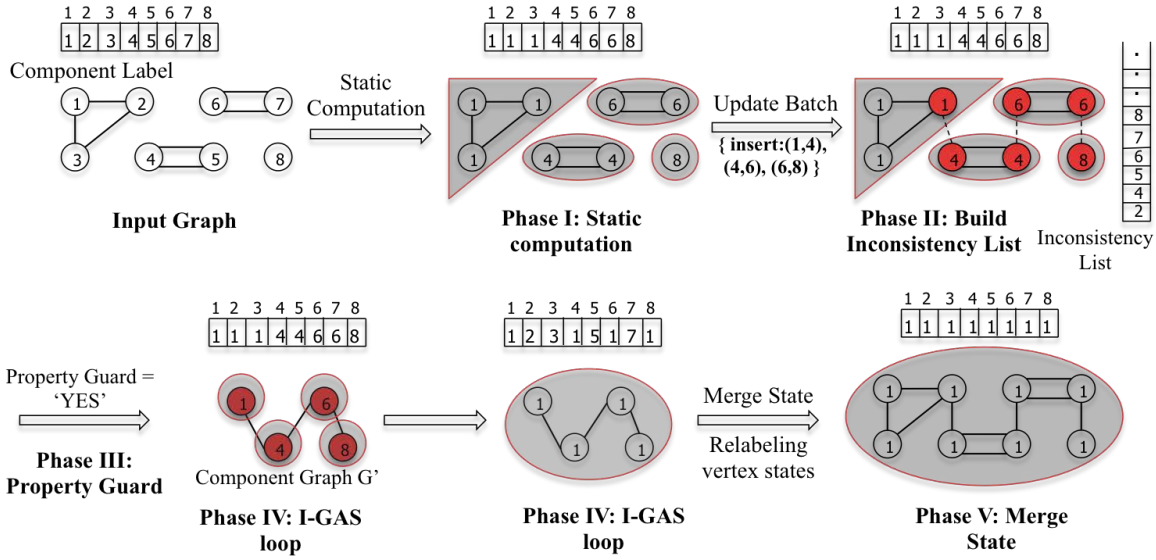

Figure 45: Computation phases of incremental connected component (CC) implemented in EvoGraph with inconsistent vertices marked red.

4.3.2 Stream Engine: Data Movement and Context Merging

The *Stream Engine (SE)* is mainly responsible for: (i) efficient asynchronous data transfer between host and GPU, and (ii) context merging of static and incremental computation on the same GPU to enable multi-level GPU sharing.

For (i), SE leverages CUDA Streams, double buffering, and hardware support like *Hyper-Qs* provided by the architectures such as NVIDIA Kepler and Maxwell, to effectively overlap data streaming and computation. SE spawns separate CUDA Streams to launch multiple kernels simultaneously and transfer batches of incremental updates to the graph asynchronously, overlapping memory copies within and across computation phases. Furthermore, as shown in Figure 35(b), EvoGraph uses separate CUDA Streams to enable deep copy operations [4], [98] in order to take advantage of the large number of hardware queues offered by modern GPU architectures. This is motivated by the fact that an update batch in EvoGraph is not a single contiguous byte-array, but consists of many sub-arrays that contain edge, vertex, and vertex/edge property update information. EvoGraph exploits this by not moving the entire update batch in one copy performed by a single CUDA stream, but instead making SE to dynamically spawn multiple CUDA Streams to move these sub-arrays to GPU. The outcome is the concurrent usage of the GPU's many hardware queues, which

consequently improves the overall throughput.

As depicted in Design III of Figure 7, SE achieves (ii) by packing multiple applications' GPU contexts into a single protection domain on-the-fly to maximize GPU resource utilization and avoid core idling. When a GPU request from a graph application (static or incremental computation) arrives, SE creates a separate CUDA stream object for it by calling *cudaStreamCreate()*, the handler to which is stored in a thread local storage. Using this handler, subsequent requests from this application is dispatched over this designated stream. Upon application exit, SE tears down the stream by calling *cudaStreamDestroy()* on the stream handler. Additionally, the GPU operations that synchronize the application and the device are replaced with their CUDA stream counterparts, e.g., *cudaDeviceSynchronize()* is replaced with *cudaStreamSynchronize()*. This ensures that all the applications packed into a single GPU context associated with a particular GPU are not blocked when one of them explicitly synchronizes its host thread with the device. Next we discuss each computation phase in detail.

4.3.3 Computation Phases in EvoGraph

Phase I: Static Graph and Metadata Preprocessing. As shown in Figure 44 and Table 7, this phase has two main purposes. First, it computes the static version of the input graph based on the traditional GAS model (Section 4.1.1). Theoretically speaking, any GAS-based static graph processing frameworks can be applied here for the static computation. In this work, we chose the highly efficient GPU-based static-graph processing framework GraphReduce as our static computation engine. Second, it computes the graph property metadata, such as parent id, vertex degree, neighbors, minimum spanning tree (MST). These property metadata play an important role in processing incremental graph algorithm in the upcoming phases. For instance, Table 7 lists what metadata are required to be computed in Phase I for different incremental graph algorithms.

Phase II: Marking Out Graph Inconsistency. As illustrated in Figure 44 and 45, this is the phase where incremental graph processing begins. This phase identifies the inconsistent part of the graph, after applying the update batch, using the *build_inconsistency_list()*

function shown in Table 7. This user-defined function takes the update batch information (e.g., edge/vertex insertions and/or deletions) and the priority attribute for each vertex to build a list of inconsistent vertices. EvoGraph also provides an option for users to construct an inconsistent sub-graph G' which can be used later. Table 7 lists the action items from Phase II for different graph algorithms.

Phase III: Determine the Execution Path Through Property Checking.

As discussed in Section 4.2.3, there are classes of incremental algorithms that cause large portions of the graph to become inconsistent and hence can result in recomputation over the entire graph. For these classes, incremental processing will not achieve any performance benefit over static recomputation and may even result in performance degradation due to the overheads associated with the incremental execution. To address this, EvoGraph allows users to define a heuristic for determining which one of the two computation (incremental processing or static recomputation) should be applied. Users may select from a set of predefined graph properties (e.g., degree, neighbor info, distance, depth, etc) or define their own properties that they believe will affect the runtime of the targeted incremental algorithm. EvoGraph will then use the selected properties to decide whether to run incremental or static recomputation by calling the *property_guard()* API for each update batch. We name this *property-based dual path execution*. *property_guard()* takes four parameters: inconsistency_list, property_list, threshold_vector, and threshold_fraction. Property List defines the set of graph properties under consideration. Threshold Vector defines a set of thresholds for properties in the Property List, above (or below) which the performance of incremental processing will drastically degrade. Finally, Threshold Fraction defines the fraction of inconsistent vertices that are above (or below) the corresponding property threshold.

Using Figure 44 as an example, when running incremental BFS, the vertex depth is one of the properties defined in the property list, with the property threshold of 2 and the threshold fraction of 0.3. In this case, EvoGraph would switch to static BFS recomputation if $> 30\%$ of inconsistent vertices have BFS depth < 2. This is in line with the observation that if an update affects a large number of vertices closer to the root of BFS tree, it is better to run a full static recomputation. Note that the thresholds for these properties and the

Algorithm 1 : I-GAS Computation Loop Per Update Batch

```
1: while(!inconsistency_list.isempty()):
2:     frontier = frontier_activate(G', inconsistency_list)
3:     IGAS(G')
4:     update_inconsistency_list(G', inconsistency_list, frontier)
```

fraction of inconsistent vertices are user-tunable parameters that are algorithm and dataset dependent and require training to derive their optimal values.

Phase IV: Incremental GAS. Incremental GAS or I-GAS [99] phase ensures that only the inconsistent or affected portion of the graph is recomputed incrementally, while maintaining the vertex states for the rest to significantly reduce the overall execution time. We find it useful to introduce the term **computational frontier** to describe the number of these inconsistent/active vertices in a given iteration of a graph algorithm. The I-GAS Engine shown in Figure 43 identifies the overlap between two consecutive versions of the evolving graph and incrementally processes the graph by only operating on the new computational frontier. With the new I-GAS computation model, users can implement the I-GAS programs as well as the *frontier_activate()* and update_inconsistency_list() APIs. Algorithm 1 shows a typical I-GAS loop, which is comprised of three basic steps that are iterated over until the inconsistency list becomes empty. It starts with a set of inconsistent vertices and calls frontier_activate() to activate the next computational frontier using the vertex priority defined in *Phase II*, and then runs an I-GAS program. An I-GAS program consists of incremental versions of the Gather, Apply and Scatter functions. By default, an I-GAS program is the same as an GAS program for static execution, but users can modify it for incremental processing. Finally, the new computational frontier information is used to update the vertex inconsistency list. Figure 44 and 45 show two comprehensive examples of this phase for BFS and CC.

Phase V: State-Merging. During this phase, EvoGraph merges the updated vertex property with the previous version of the graph. Also, it decides if edge insertions or deletions are required to be applied to the recent version of the static graph G before processing the next update batch, based on algorithms' *merge patterns*. Table 7 shows three graph algorithms that belong to three different classes of merge patterns: Stateful

(Breadth-First Search), Partially Stateless (Connected Components) and Fully Stateless (Triangle Counting). We summarize these three merge patterns as follows:

- **Stateful**: This type of incremental algorithms typically operate on the graph properties that have global effects, and must apply all the updates (both insertions and deletions) of the current batch to G at the end of the I-GAS loop. For example, vertex depth calculation in BFS requires consideration of any added/deleted edges.

- **Partially Stateless**: In each incremental iteration, this type of algorithms have dependency on either deletions or insertions from the previous update batch, but not both. In other words, either deletions or insertions are required to be merged with G at the end of the I-GAS loop. Hence the rest of the updates, lacking dependency, can be processed anytime during the execution without influencing the final result and their merger with G is deferred by EvoGraph. Connected Components belongs to this category.

- **Fully Stateless**: This category of incremental algorithms update the graph properties that only have local effects. More specifically, neither insertions nor deletions within each incremental iteration have any dependency on the previous update batch. Therefore, both insertions and deletions are deferred by EvoGraph. Triangle Counting shown in Table 7 belongs to this category. Other examples include clustering coefficients and vertex degree counting.

Next, we will showcase the implementation of incremental BFS, CC and TC belonging to these three classes of incremental algorithms in EvoGraph.

4.4 Case Studies

Stateful. Detailed in Table 7, BFS is an example of a Stateful algorithm as it requires all the updates from one batch to be merged with the original graph before processing the next batch. Figure 44 illustrates the five computation phases of an incremental BFS. Phase I involves static computation of BFS depths from the source vertex (handled by GraphReduce [98]) and metadata computation of properties such as degree and parent

```
function meta_computation():
//Compute the degree and parent id for each vertex

function build_inconsistency_list (UpdateList edge_list
,Graph G, Priority MIN_PRIORITY):
min_depth = MAX_INT
for each e in edge_list
   for each in-edge e' with e'.dst = e. dst
      min_depth = min(VertexProperty[e'.src].depth
     ,min_depth)
   current_depth = VertexProperty[e.dest].depth
   if(current_depth > min_depth + 1)
      VertexState[e.dst].depth =  min_depth + 1
      inconsistency_list =
      inconsistency_list U {e.dst,MIN_PRIORITY_QUEUE}
G' = G
return (inconsistency_list,G')

function property_guard(DepthThreshold DP
,ThresholdFraction f):
if fraction of inconsistent vertices
        with (depth < DP) > f
  run static re-computation
else run incremental algorithm

function frontier_activate(Graph G'
,List inconsistency_list):
//Extract all duplicate minimums
v_min_set = inconsistency_list.extract_min()
Activate(v_min_set)

function update_inconsistency_list(Graph G'
,List inconsistency_list,Set frontier):
//Similar to build_inconsistency_list() but checks for
//consistency of all the successors of the frontier

//I-GAS computation loop
function I-GAS(List inconsistency_list,Graph G'):
while(!inconsistency_list.empty())
  frontier = frontier_activate(G',inconsistency_list)
  IGAS(G')
  update_inconsistency_list(G',inconsistency_list
  ,frontier)

function merge_state(Graph G,Graph G')
// Do nothing in BFS
```

Figure 46: Stateful example: Implementation of incremental BFS using EvoGraph APIs.

vertex id information for each vertex in the graph. In Phase II, vertices that have incorrect depth values [88] after applying the current update batch are marked as inconsistent and

added to a container with min-priority that is ordered by depth value. Phase III checks for any listed property to decide if the framework should run the incremental version or re-run the static recomputation algorithm. In BFS, we use vertex depth as the property and check if the fraction of inconsistent vertices with depth values below a certain threshold (e.g., 2) have crossed certain limit. In Phase IV, EvoGraph fixes the inconsistency in the vertices of the graph in the order of their minimum depth values, as described in the algorithm by Ramalingam and Reps [88]. Therefore, in each iteration of I-GAS loop, inconsistent vertices with the minimum depth value (can be multiple vertices) are activated and made consistent; and the inconsistency list is updated. Phase V is trivial as the vertex states are shared and hence do not require merging. Figure 46 shows the BFS implementation in EvoGraph.

Partially Stateless. Shown in Table 7 and Figure 45, Connected Components (CC) is a partially stateless algorithm because only deletions are required to be merged with G. For deletions we need to re-run the static algorithm and there are few proposed optimizations [48], [81] to eliminate false delections so that a component will not be broken. Phase I calculates the static version of connected component. In Phase II, EvoGraph builds the inconsistent graph G' with vertex ids as the component labels in the original graph, and for each edge insertion in G it adds an edge in G' if the endpoints of the edge belong to different components. G' is also known as *component graph* [48]. Phase III checks for the fraction of inconsistent vertices that belong to disjoint components. Phase IV runs static connected components algorithm on G'. Note that EvoGraph has successfully reduced the incremental problem in G to a static problem in G'. Finally, Phase V relabels the vertices in G from the computed component labels in G'. Figure 47 shows the CC implementation in EvoGraph.

Fully Stateless. As illustrated in Table 7, Triangle Counting (TC), which measures the total number of closed triangles in a graph representing small-worldness of a graph, is a fully stateless algorithm because both insertions and deletions from an update batch are not required to be merged with the original graph before processing the next batch. Phase I computes the static version of the algorithm and the degree property for each vertex (metadata computation). In Phase II, EvoGraph marks the endpoints of every

```
1 function meta_computation():
2 //Compute the degree for each vertex
3
4 function build_inconsistency_list(edge_list,
5 G, DEFAULT_PRIORITY):
6 //For each edge inserted, add an edge to G'
7 //if endpoints belong to different components
8 for each edge e in edge_list
9   label1 = VertexProperty[e.src].component_label
10  label2 = VertexProperty[e.dst].component_label
11  if(label1 != label2)
12    G' = G' U (label1, label2)
13    inconsistency_list=
14    inconsistency_list U {e.src,e.dst,DEFAULT_PRIORITY}
15 return (inconsistency_list,G')
16
17 function property_guard(degree: DE,
18 disjoint component: count threshold_fraction f):
19    if fraction of inconsistent vertices with
20    (degree > DE or count > n*f) > f
21       run static re-computation
22    else run incremental algorithm
23
24 function frontier_activate(G',inconsistency_list):
25 // Using extract operation on inconsistency_list
26 for every edge e in G'
27   activate(e.src)
28   activate(e.dst)
29
30 function update_inconsistency_list(G',
31 inconsistency_list, new_inconsistency=NULL):
32 if(G.activity.empty())
33    inconsistency_list.clear()
34
35 //I-GAS computation loop
36 function I-GAS(inconsistency_list,G'):
37 While(!inconsistency_list.empty())
38   if(itr=1)
39      frontier_activate(G',inconsistency_list)
40   else frontier_activate(G',NULL)
41   Modified-GAS(G')
42   update_inconsistency_list(G',inconsistency_list)
43
44 function merge_state():
45  // relabel the vertex component ids
```

Figure 47: Partially Stateless example: Implementation of incremental Connected Components using EvoGraph APIs.

edge inserted or deleted and their respective neighboring vertices as inconsistent. Then it builds the inconsistent graph G' with edges incident on every inconsistent vertex. Phase III checks for the fraction of inconsistent vertices in G that have degree above certain

threshold. Phase IV is similar to that in CC, activating all the vertices in G' and then running the static algorithm on G'. Note that EvoGraph has again successfully reduced a fully dynamic (having both insertions and deletions) problem in G to a static problem in G'. Finally, Phase V updates the triangle counts and the degree info in G using the corresponding computed values in G'. Users can also implement a Bloom Filter version of the incremental algorithm for fast membership queries as shown in [46]. Figure 48 shows the TC implementation in EvoGraph. Note that although updates of the current batch are not required to be merged to the original graph for its processing, they might be needed for processing of future updates. Therefore, EvoGraph simultaneously merges the updates while processing an update batch. This is still a fully stateless algorithm as the merge step doesn't come in the critical path of incremental processing of the current update batch.

4.5 Experimental Evaluation

4.5.1 Experimental Setup

Evaluation Platform. We evaluate EvoGraph on a typical heterogeneous HPC node equipped with 12-core Intel Xeon X5660 processors running at 2.8 GHz with 12 GB of DDR3 RAM, and one attached NVIDIA Tesla K40c GPU with 15 SMX multiprocessors and 12 GB GDDR5 RAM. The Kepler GPU is enabled with CUDA 7.0 runtime and the version 352.79 driver, while the host CPU is running Fedora version 20 with kernel v.3.11.10-301 x86. We use GraphReduce [98] and STINGER [47], [48], [46] for performance comparisons. All the runs are compiled with the highest optimization level flag. Updates are provided in batches to EvoGraph and STINGER where each batch size can range from 100,000 up to one million. For all three algorithms, the batch consists of 99% edge insertions and 1% deletions. The endpoints of the edges used for batch updates are generated randomly.

Graph Dataset. For evaluating the performance of EvoGraph, we use a mix of real-world and synthetic datasets. Their graph properties are shown in Table 8. The five real-world datasets are from University of Florida Sparse Matrix Collection [22]. The synthetic datasets are obtained from the Graph500 RMAT data generator [84] using scale 19, 20 and 21 with average degree of 16 per vertex, labeled as G19D16, G20D16, and G21D16

```
function build_inconsistency_list(edge_list,G,property):
//insertions
for each e in edge_list
  VertexProperty[e.src].tri_count = 0
  VertexProperty[e.dst].tri_count = 0
  inconsistency_list =
  inconsistency_list U{e.src,e.dst,DEFAULT_PRIORITY}
  G'= G' U (e.src, e.dst)
  for each neighbor x of e.src
     inconsistency_list =
     inconsistency_list U{x,DEFAULT_PRIORITY}
     G' = G' U (e.src, x)
  for each neighbor y of e.src
     inconsistency_list =
     inconsistency_list U {y,DEFAULT_PRIORITY}
     G' = G' U (e.dst,y)
return (inconsistency_list,G')

function property_guard(degree: DE,threshold_fraction f):
if fraction of inconsistent vertices with
(degree > DE > n*f) > f
   run static re-computation
 else run incremental algorithm

function frontier_activate(G',inconsistency_list):
// Using extract operation on inconsistency_list
for every edge e in G'
  activate(e.src)
  activate(e.dst)

function update_inconsistency_list(G',inconsistency_list,
new_inconsistency=NULL)
if(G.activity.empty())
   inconsistency_list.clear()

//I-GAS computation loop
function I-GAS(inconsistency_list,G'):
While(!inconsistency_list.empty())
  if(itr = 1)
     frontier_activate(G',inconsistency_list)
  Modified-GAS(G')
  update_inconsistency_list(G',inconsistency_list)

function merge_state(inconsistency_list) :
for each v in inconsistency_list
  VertexProperty[v].tri_count += tri_count_old
```

Figure 48: Fully Stateless example: Implementation of incremental Triangle Counting using EvoGraph APIs.

respectively.

Table 8: Graph Datasets Under Evaluation

Graph Dataset	Type	#Vertices	#Edges
hollywood-2009	real world	1,139,905	113,891,327
indochina-2004	real world	7,414,866	194,109,311
ljournal-2008	real world	5,363,260	79,023,142
kron_g500-logn21	real world	2,097,152	182,082,942
uk-2002	real world	18,520,486	298,113,762
G19D16	synthetic	524,288	8,388,608
G20D16	synthetic	1,048,576	15,700,394
G21D16	synthetic	2,097,152	31,771,509

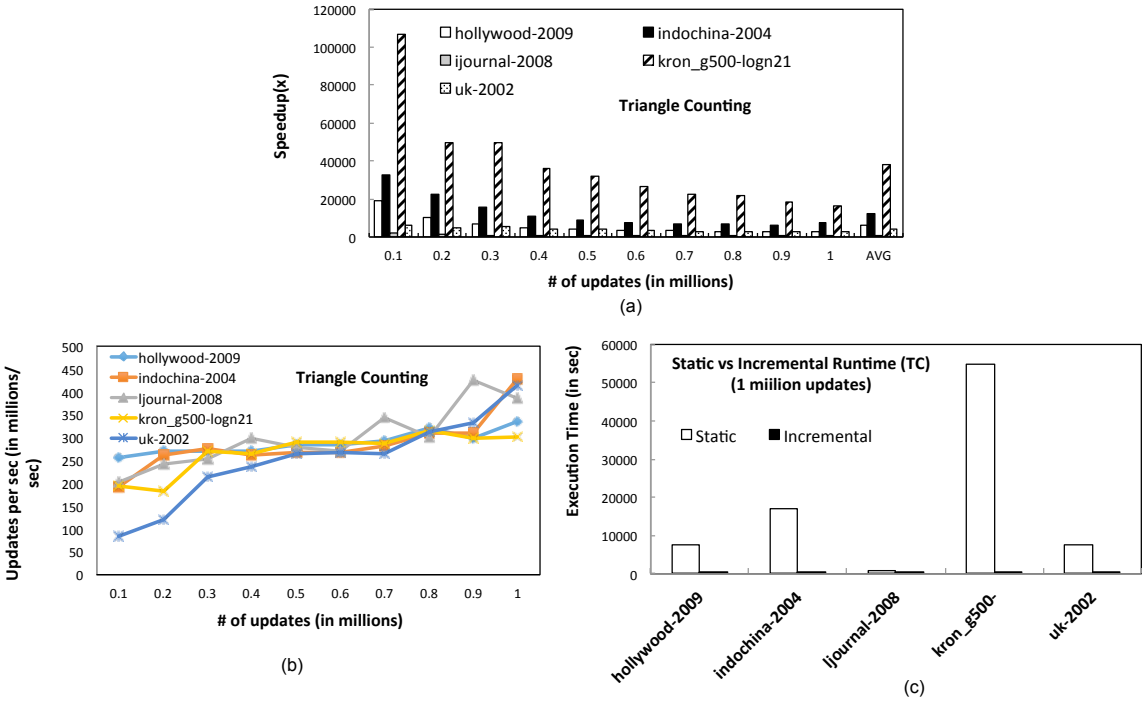

Figure 49: Triangle Counting (TC): (a) EvoGraph's speedup over the static computation using GraphReduce; (b) Update Rate that EvoGraph achieves; (c) For 1 million updates, EvoGraph vs. Static Runtime using GraphReduce.

Evaluated Algorithms. Three widely used graph algorithms are evaluated, including Triangle Counting (fully-stateless), Connected Components (partially-stateless) and Breadth-First Search (stateful), to cover the three classes of algorithm patterns (Section 4.4). Algorithms requiring undirected graphs as inputs (e.g., CC) are stored as pairs of directed edges.

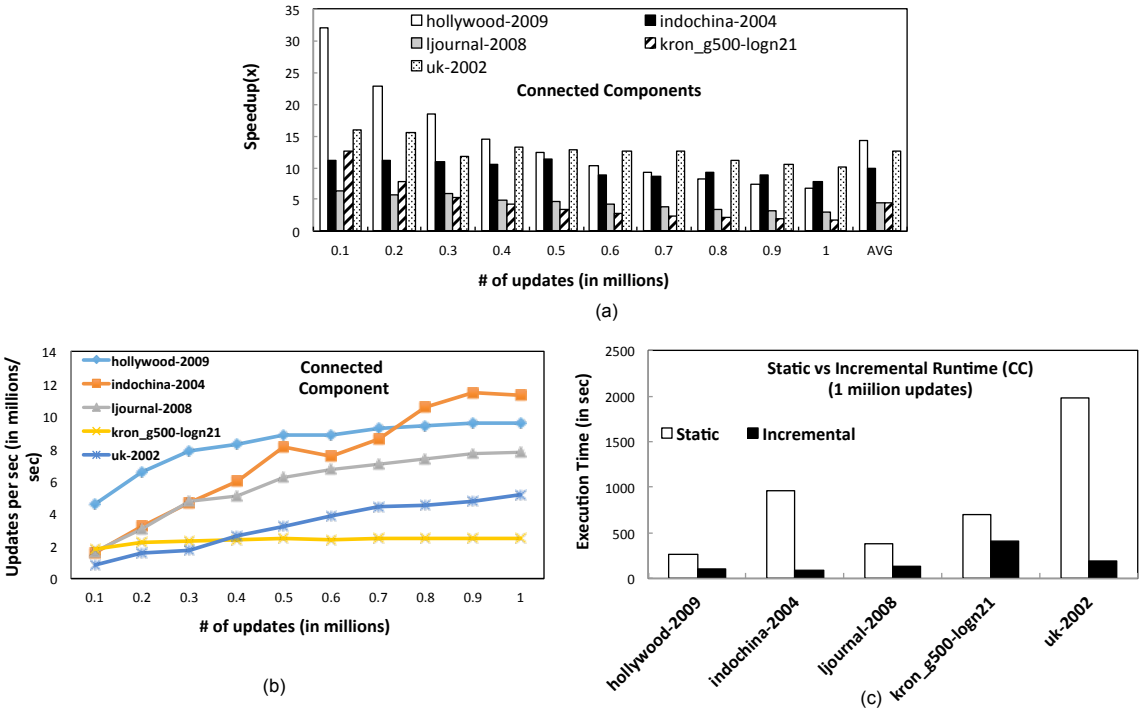

Figure 50: Connected Components (CC): (a) EvoGraph's speedup over the static computation using GraphReduce; (b) Update Rate that EvoGraph achieves; (c) For 1 million updates, EvoGraph vs. Static Runtime using GraphReduce.

4.5.2 EvoGraph Vs. Static Recomputation

To process evolving graphs, the state-of-the-art GPU frameworks suchs as GraphReduce and Cusha, which only process static graphs, have to follow a store-and-static-compute model, repeatedly running static graph computation on the snapshots of the evolving graph. Here we showcase the benefits of incremental graph analytics (EvoGraph) over such offline static recomputation using GraphReduce [98]. Figure 49(a) , 50(a) and 51(a) show that EvoGraph achieves an average performance improvement of 12278x, 9.13x and 1.16x over GraphReduce across all the datasets for TC, CC and BFS, respectively, with the update batch size going as high as 1 million updates. Figure 49(b), 50(b) and 51(b) demonstrates that EvoGraph is able to achieve up to 429 million updates/sec. Figure 49(c), 50(c) and 51(c) compares the incremental and static runtime performance for TC, CC and BFS for a batch size of 1 million updates. It clearly demonstrates the performance benefit using EvoGraph for incremental graph processing. The large performance improvement achieved

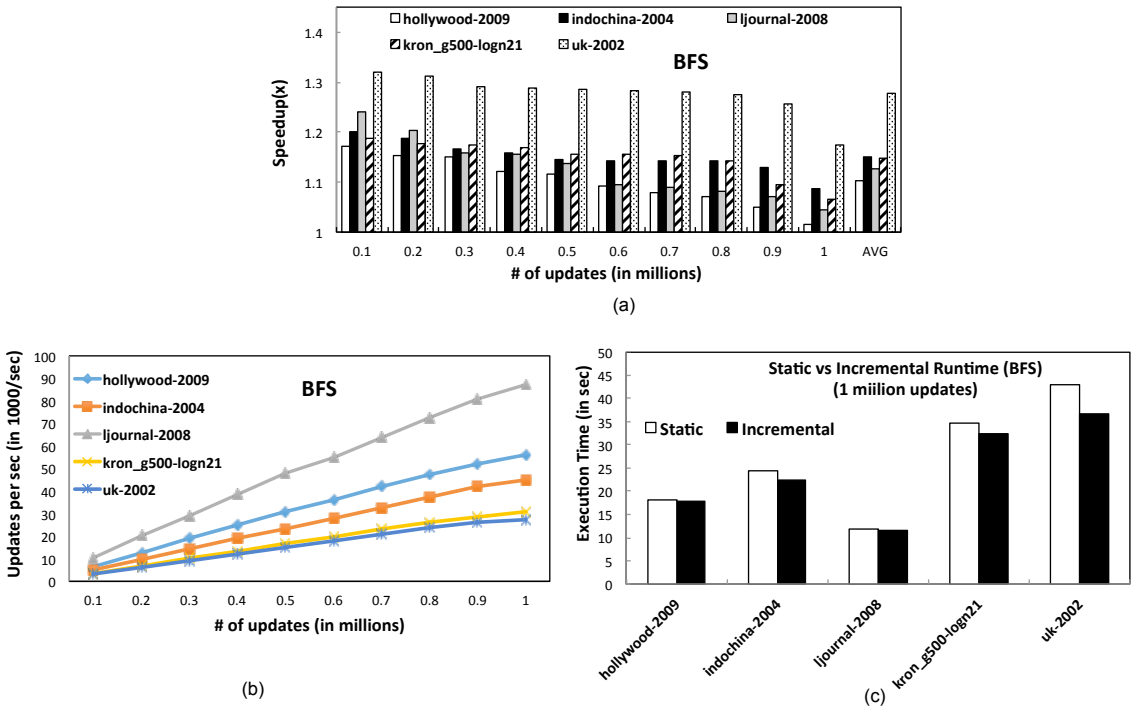

Figure 51: Breadth First Search (BFS): (a) EvoGraph's speedup over the static computation using GraphReduce; (b) Update Rate that EvoGraph achieves; (c) For 1 million updates, EvoGraph vs. Static Runtime using GraphReduce.

by EvoGraph is due to (i) the use of incremental computation in the I-GAS execution model to compute the vertex states for only the inconsistent set of vertices, as opposed to executing the algorithm on the entire input graph; (ii) asynchronous mode and deep copy operations between host and device leveraging CUDA Streams and *Hyper-Qs* to keep both compute and memory-copy engines occupied simultaneously; (iii) concurrent static and incremental graph processing via time and space sharing on GPU by Context Merging (Section 4.2.4), resulting in substantial reduction in context-switching overhead and GPU core idling. Furthermore, we can draw key inferences as to how the performance of incremental execution varies with algorithm type, update batch size and input graph size.

4.5.3 Sensitivity Analysis

Effect of graph algorithm. Figure 49, 50 and 51 show that the maximum speedup achieved by EvoGraph over static recomputation occurs with TC (fully stateless), followed by CC (partially stateless) and then BFS (stateful). Also, the relative runtime performance

of TC is the highest compared to CC and BFS. Additionally, the average system throughputs achieved across all graph algorithms for a batch size of 1 million updates are 372 million, 7.3 million and 50K updates/sec for TC, CC and BFS, respectively. This substantial difference in performance across different merge patterns is because the fraction of the graph that becomes inconsistent after applying an update batch as the I-GAS loop unfolds increases in the order of fully-stateless, partially-stateless, and stateful. In other words, fully-stateless or partially-stateless algorithms only affect the graph locally, so the incremental runtime is bounded by the size of the update batch. On the contrary, stateful algorithms like BFS calculate a global property (e.g., vertex depth), so the incremental computation affects a larger portion of the graph and hence achieves lower speedup. Furthermore, during the State-Merging phase of incremental BFS, the new update batch is applied or merged with the current static version which in turn is copied back to the GPU. This incurs a large data transfer overhead, whereas for TC and CC the data transfers are of the order of the update batch size, keeping the memcpy time small.

Effect of update batch size. Shown in Figure 49(a)-51(a), the speedup falls as the update batch size increases because of the increasing problem size. We also observe that both the runtime (because of decreasing speedup) and update rate increases with the batch size (Figure 49(a)-51(a)), which implies that the decrease in speedup changes slower with respect to dramatic update rate increases for larger batches.

Effect of input graph size. Finally, from Figure 49(b)-51(b) we can observe that BFS's throughput increases with the batch size a lot faster for smaller graphs (e.g., ijournal-2008) than for the larger ones (e.g., uk-2002). This happens because for updates affecting a particular BFS level, the cost of incremental computation increases with the size of the inconsistent subgraph which in turn is proportional to the size of the input graph. On the other hand, the update rates of CC and TC show no correlation with the input size as the graph properties under consideration are local or semi-local, whose performance is more likely dependent on properties such as number of disjoint components and vertex degree.

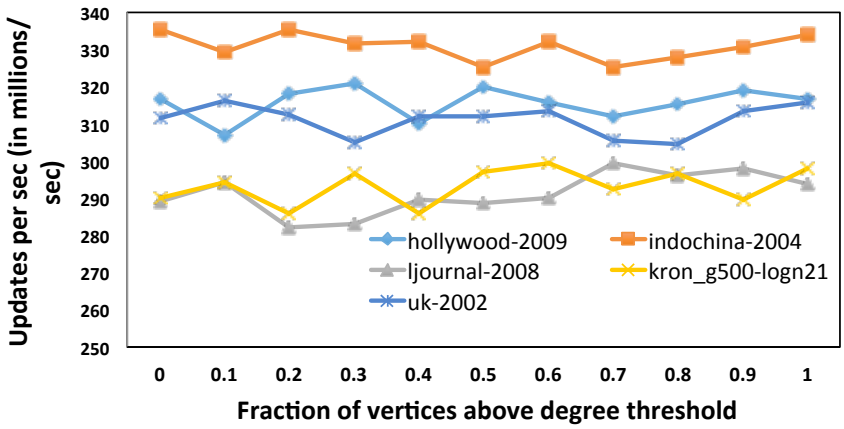

Figure 52: Impact of vertex degree property on the update rate of Triangle Counting algorithm.

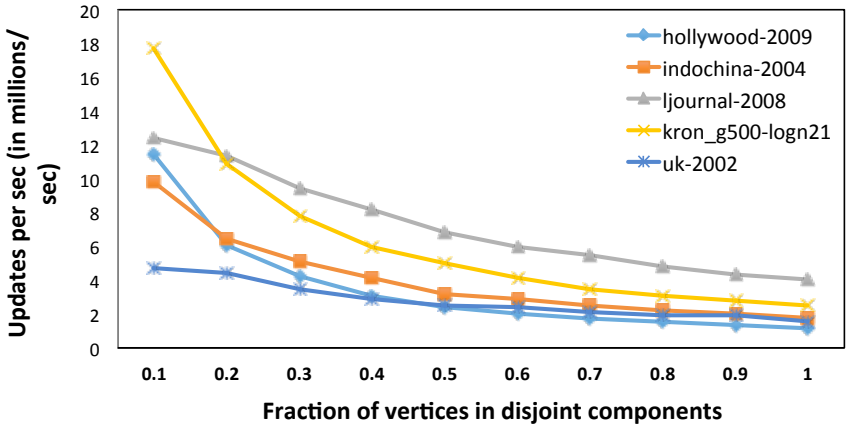

Figure 53: Impact of disjoint components property on the update rate of Connected Components algorithm.

4.5.4 Performance Implications of Graph Properties

Now we evaluate how properties of the inconsistent vertices from a given update batch affect the average runtime and throughput of incremental graph processing. For demonstration, we have chosen specific graph property for each of the three algorithms: vertex degree for TC, vertices with disjoint components for CC, and vertex depth from the source for BFS. The rationale behind choosing these properties is that they play an essential role in the static graph algorithms that we are comparing against.

Triangle Counting (Vertex Degree): Figure 52 shows the change in the update rate for TC versus the fraction of updates (insertions and deletions) affecting vertices with

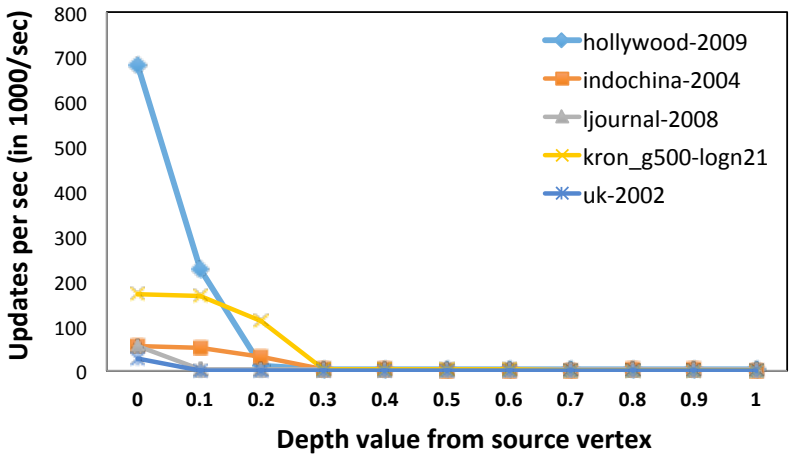

Figure 54: Impact of vertex depth property on the update rate of Breadth First Search algorithm.

degree greater than a certain threshold (e.g. 1900 for Hollywood graph). We vary the fraction of inconsistent vertices with degree higher than the threshold degree in a given update batch and then evaluate its effect on the incremental runtime and the update rate. We can observe that the degree property does not have a dramatic effect on the update rate for TC. This is because TC is a stateless algorithm for both insertions and deletions to the graph, and the size of the sub-graph G' created in Phase II (see Table 7), and thus the incremental runtime is independent of the vertex degree. Therefore, the update rate remains relatively constant even when more edges are inserted and/or deleted near high degree vertices or supernodes.

Connected Components (Disjoint Components): Figure 53 shows that the update rate for CC decreases as the fraction of edges inserted (whose endpoints belong to different components in the original graph) is increased, with a maximum slowdown of 10.2x across all the datasets. This happens because the endpoints of an edge falling in the same component results in a self-edge in the component graph and ignored by EvoGraph. On the contrary, if the endpoints are in different components implying that there is a corresponding edge in the component graph G'. From Table 7, EvoGraph reduces incremental CC on G to static CC processing on G' and increasing the number of vertices with disjoint components proportionately increases the size of G', and subsequently increasing the incremental

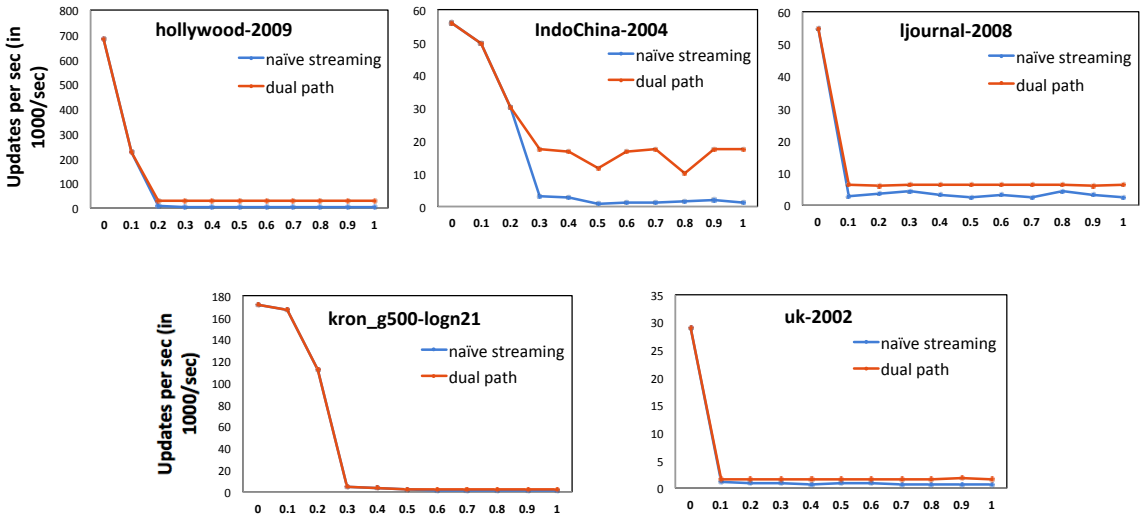

Figure 55: Property-guard heuristic vs. naive streaming in incremental BFS using vertex depth property for five graph inputs. The x-axis represents the fraction of vertices below depth threshold of MAX_DEPTH/4.

processing time.

BFS (Vertex Depth): Figure 54 shows that increasing the fraction of vertices with depth below a given threshold (MAX_DEPTH/4 in our case) causes a sharp decline in the update rate. This slowdown comes from more insertions and deletions on these lower-depth vertices closer to the root vertex, which results in the I-GAS loop making a much larger portion of the graph inconsistent with each increment. The maximum slowdown (max to min ratio of the update rate) across all datasets is 213x which can lead to dramatic system performance degradation and hence motivates our *property guard* heuristic which we discuss next.

Property-Guard Heuristic: In these sets of experiments we show how EvoGraph uses property information and adapts to situations where the incremental processing performs worse than static recomputation. Figure 55 shows that the performance of incremental BFS for naive streaming without considering the property information of the current update batch falls relative to static processing beyond a threshold fraction of vertices with depth threshold below MAX_DEPTH/4. For Hollywood, Indochina, Ljournal, KronLogn21 and UK-2002 this threshold fractions are 0.2, 0.3, 0.1, 0.5 and 0.1 respectively. As discussed in the previous section, this degradation in incremental performance is because a larger number

of updates to these lower-depth vertices results in a large portion of the graph becoming inconsistent and hence significant increase in processing time. In phase III, EvoGraph analyzes the current update batch for the depth threshold and if the batch has a fraction of vertices beyond certain thresholds, instead of proceeding to I-GAS incremental execution, processes the update batch with static recomputation ensuring the worst-case performance has the same lower bound as static recomputation. We achieve a maximum speedup of 18.4x, using this heuristic, compared to a naive streaming approach (Indochina).

4.5.5 EvoGraph VS. STINGER

Figure 56 shows the comparison between the update rate for EvoGraph vs. STINGER [29] for TC and CC. Note that for fairness data transfer time between host and GPU has been included for EvoGraph computation while STINGER is not subject to such overhead. Across all 3 synthetic datasets STINGER shows a max update rate of 2.1 million versus 488 million updates/sec with EvoGraph for the S19D16 case, a 232.4x increase in throughput. EvoGraph also shows better scalability as batch size increases because of (1) the massive parallelism offered by thousands of cores of GPU that is sufficient to overcome the substantial overheads from the large data movement over PCIe, (2) EvoGraphs hybrid data structure of edge-list for incremental updates and compressed matrix format for static versions of the graph. STINGER uses edge-list based data structures for both the static and incremental graph processing, which results in faster data structure update time but slower traversal time due to list traversal. EvoGraphs hybrid data structure thus enables faster updates (via the edge-list) as well as fast static computation (via compressed matrix format).

4.5.6 Discussion

Experiments demonstrate that (1) EvoGraphs incremental approach to process time-evolving graphs on GPUs, combined with its asynchronous and deep memory copy operations on separate CUDA streams leveraging the multiple hardware queues (Hyper-Qs) in GPUs; and

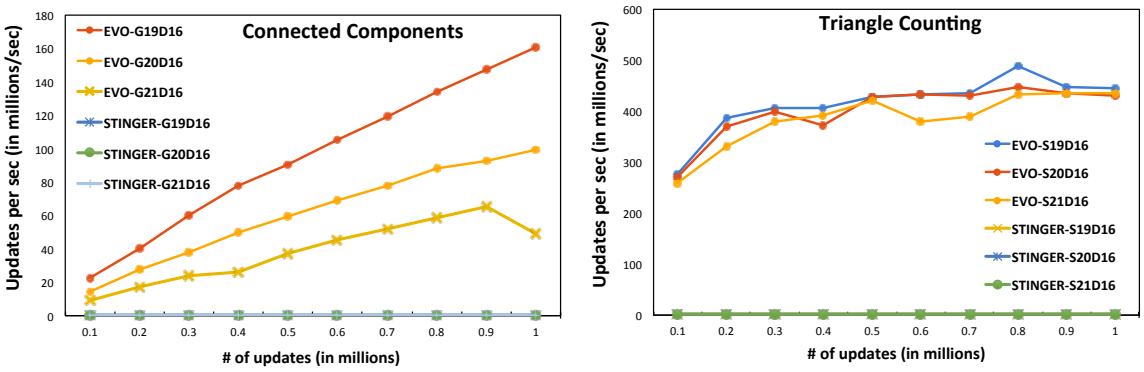

Figure 56: EvoGraph vs STINGER throughput comparison for (a) Connected Components and (b) Triangle Counting.

context merging of various static and incremental graph algorithm for better GPU utilization, can achieve dramatic speedups of up to 12278x over traditional store-and-static-recompute model with a maximum system throughput of 429 million updates/sec across several real-world and synthetic graphs. (2) The graph algorithm type, based on their merge pattern, can have a dramatic impact on the update rate achieved with a fully stateless (TC) and stateful (BFS) algorithm achieving an average throughput of 50K and 372 million updates/sec respectively with a batch of 1 million updates. (3) Vertex properties of an update batch investigated either do not affect the update rate, as in the case of vertex degree with TC, or can have a dramatic impact on the update rate, as in the case of vertex depth with BFS resulting in a maximum slowdown of 213x. The impact of such properties largely is related to the portion of the graph that is made inconsistent with each iteration. (4) The property-based dual path execution optimization in EvoGraph to choose between an incremental vs static run over a particular update batch can achieve up to 18.4x compared to naive streaming approach. (5) Leveraging the massive parallelism offered by thousands of cores of GPU and its hybrid data structure of edge-list for incremental updates and compressed matrix format for static versions, EvoGraph achieves a speedup of upto 232.4x compared to STINGER.

4.6 Chapter Summary

This chapter presents EvoGraph, an accelerator-based high-performance incremental graph processing framework for processing time-evolving graphs. Technical advances offered by EvoGraph include: (1) an incremental variant of Gather-Apply-Scatter called I-GAS to compute graph properties only for the inconsistent subgraphs, (2) a user-tunable property-based optimization called property-guard for switching between I-GAS and static recomputation, and 3) deep memory copy operations via separate CUDA Streams and GPU context merging for improved asynchronous computation and communication performance. Evaluation on a variety of graph inputs and algorithms demonstrates that EvoGraph achieves a system throughput of up to 429 million updates/sec and a 232x speedup when compared to competitive frameworks like STINGER. Furthermore, the property-guard optimization of EvoGraph achieves a speed up of up to 18.4x over a naive streaming approach. Future work will look at extending EvoGraph to multi-node clusters and extreme-scale datasets.

CHAPTER V

RELATED WORK

5.1 Accelerator-based Graph Processing

Merrill *et al.*[82] present a parallelization of BFS tailored to the GPU's requirement for large amounts of fine-grained BSP; they achieve an asymptotically optimal $O(|V| + |E|)$ work complexity. Hong *et al.*[63] propose warp-level load-balancing that defers outliers and performs dynamic workload distribution to speed up graph algorithms through heavy-weight atomic operations on global memory. Duong *et al.* [45] conduct detailed GPU-based optimizations for PageRank and achieves significant speedup over a multi-core CPU implementation. Chapuis *et al.* [43] provide an algorithmic optimization solution to speedup all-pairs shortestpath (APSP) for planar graphs that exploits the massive on-chip parallelism available on GPUs. The GraphReduce [95] framework can be extended to implement the algorithm-specific optimizations above and in contrast to such work, it offers user-level APIs for programming graph algorithms and provides a general framework addressing a wide range of parallel graph algorithms and hiding architecture-level optimizations from users.

Concerning frameworks for GPU-based graph processing, earlier work like *Medusha* [119] introduces some basic graph-centric optimizations for GPUs, offering a small set of user-defined APIs, but its performance is not comparable to the state-of-the-art low-level GPU optimizations. To address this issue, *MapGraph* [52] and *VertexAPI*[23] implement runtime-based programming frameworks with levels of performance that match those seen for low-level specific algorithm optimizations. *MapGraph* chooses among different scheduling strategies, depending on the size of the frontier and the adjacency lists for the vertices in the frontier. It also uses a Structure Of Arrays (SOA) pattern to ensure coalesced memory access. *VertexAPI* provides a GAS model-based GPU library, gaining high performance

primarily from using the ModernGPU [10] library for load balancing and memory coalescing. *CuSha* [67] identifies the shortcomings of the state-of-the-art CSR-based virtual warp-centric method for processing graphs on GPUs and in response, proposes G-Shards and Concatenated Windows to address its performance inefficiency. All of the approaches above make the fundamental assumption that large graphs fit into GPU memory, a restriction that is not present for GraphReduce. As discussed in Chapter 3, GraphReduce not only addresses the processing of out-of-memory graphs, but also matches the in-memory performance seen with these state-of-the-art approaches, in many cases outperforming them significantly.

5.2 Out-of-Core Graph Processing

Out-of-Core graph processing has been concerned with CPU-based hosts processing graphs that do no fit into host memory. *GraphChi* [71], for instance, is based on a vertex-centric implementation of graph algorithms where graphs are sharded onto the SSD drives attached to the host. Its SSD-targeting sharding methods motivate GraphReduce's approach to how GPUs view and interact with host memory. (Table 4). GraphChi proposed the concept of partitioning a graph into shards and a novel parallel sliding windows technique to load a subgraph into the CPU memory. This method enables a sequential access of memory as the in-edges are sorted according to their source vertices. We also borrow from X-Stream [93] the edge-centric way to organize data for our GAS model. To improve GraphChi with the scenario that large graphs commonly have more edges than vertices, *X-Stream* [93] enables an edge-centric scatter-gather model. Unlike GraphChi which requires pre-processing in the form of sorting the in-edges, X-Stream streams unordered edge lists and puts the updates into buckets corresponding to different vertex intervals. Both Graphchi and X-Stream are CPU-based implementations. Although they both have multi-threaded version, they do not come close to the parallelism offered in GPUs which our GraphReduce framework takes advantage of. Discussed in Chapter 3, GraphReduce enables a hybrid programming model and significantly outperforms state-of-the-art X-Stream for different graph inputs processed by various algorithms. *Totem* [53] offers a high-level abstraction for graph processing on

GPU-based systems, by statically partitioning graphs into GPU and host memories, placing low-degree vertices on the host and high-degree vertices on the GPU. The approach improves performance if the graphs follow a power-law vertex degree distribution, and as graph size increases, only a fixed sub-graph able to fit in GPU memory will be processed, resulting in GPU underutilization and eventual CPU-based bottlenecks for graph processing. *Green-Marl* [62] is a Domain Specific Language (DSL) for efficient graph analysis on CPUs; its implementation is not amenable to many-core architectures. Also, it requires static analysis to generate thread assignment which will not work for GPU runtime.

5.3 Distributed graph processing

This involves processing of large-scale graphs in a distributed fashion by making use of the combined memories of multiple machines to fit large graphs that don't fit in a single machine. Pregel [80] provides a synchronous vertex-centric graph processing framework that is based on message passing. GraphLab [77] provides a framework for machine learning and data mining while PowerGraph [56] exploits the power-law vertex degree distribution for efficient data placement and computation. ASPIRE [114] adopts an asynchronous mode of execution with a relaxed consistency to improve the remote access latency. These project are complimentary to EvoGraph, in that they could be used to implement the static component of EvoGraph computation while I-GAS can be leveraged to make these distributed frameworks more dynamic.

5.4 Dynamic graph processing

There are two broad categories of dynamic graphs processing (1) Offline processing of dynamic graphs that involves the generation, storing, and analysis of a sequence of versions or time-stamped snapshots of dynamic graphs for the calculation of some global graph property. (2) Online processing of dynamic graphs that involve real-time, continuous query processing over streaming updates on the evolving graph. EvoGraph is a framework designed to address the later problem. Chronos [61], GraphScope [109], and TEG [50] are some examples of the most recent work in offline dynamic graph processing. Chronos is a high-performance system that supports incremental processing on temporal graphs using a

graph representation that places graph vertex data from different versions together leading to good cache locality. GraphScope proposes encoding for evolving graphs for community discovery and anomaly detection.

5.5 Real-time, continuous query processing

This implies certain memory constraints that might not allow keeping multiple versions of the evolving graph. STINGER [47] defines an efficient data structure to represent streaming graphs that enables fast, real-time insertions and/or deletions to the graph. Several applications have been built using the STINGER graph representation like clustering coefficient [3] and connected components [4]. Unlike STINGER which uses a single data structure for both static and dynamic graph analysis, EvoGraph uses a novel hybrid data structure that allows for incremental computation on edge lists and a compressed format for static graph computation. A key takeaway of these related work is that any existing algorithm that can be reduced to a GAS-based graph sub-problem can easily leverage EvoGraph to implement their incremental versions.

CHAPTER VI

CONCLUSIONS AND FUTURE DIRECTIONS

In this thesis we have presented and evaluated system design principles for efficient scheduling and resource management in heterogeneous, accelerator-based systems with 1000s of compute cores and complicated memory hierarchy in order to achieve better core utilization, efficient data movement and seamlessly scaling to large input datasets, particularly those arising from the processing complex GPU-based applications. The proposed technologies address this resource management and scheduling challenges by bringing support for load balanced scheduling of application requests to avoid request collisions, feedback-based mechanisms for efficient data movement and placement, system-level support for reducing core idling and seamlessly scaling to large input datasets for out-of-core processing to achieve optimal performance in a wide variety of applications.

We presented Strings, a cluster-wide GPU aggregation and two-level hierarchical scheduling infrastructure that decomposes the problem of GPGPU request scheduling into a novel combination of workload balancing and device-level scheduling. To achieve high GPU utilization and minimize context switching overhead, it provides system-level support to dynamically encapsulate the GPU contexts of multiple applications into a single umbrella context. Further dynamic policy switching based on device-level scheduler feedback and advanced scheduling policies like Phase Selection (PS) that aims to keep all of a GPUs hardware units busy by intelligently selecting applications operating in different phases of their use of the GPU, achieve high system throughput without compromising with the fairness among multi-tenant applications. Across a wide variety of workloads and system configurations, Strings achieves substantial throughput and fairness improvement compared to the CUDA runtime and competing GPU scheduling solutions.

This dissertation addresses the technical challenges in large-scale graph analytics including dealing with the dynamic nature of graph parallelism, coping with constrained on-GPU

memory capacity and addressing programmability issues for developers with limited insights into how to best exploit the resources of evolving and varied GPU architectures. Towards this end, we have developed GraphReduce framework that enables processing of graphs with memory footprint much exceeding that of GPU memory, by sharding graph data and asynchronously moving shards between GPU and host memories. Technical advances offered by GraphReduce include its usage of a hybrid programming model of edge- and vertex-centric processing, asynchronous execution/spray operation, dynamic phase fusion/elimination, and dynamic frontier management. With these optimizations, GraphReduce achieves significant performance improvement over competing out-of-core implementations of graph processing for several real-world large-scale graphs processed by various algorithms.

To address the problem of anlayzing dynamic graphs that are changing over time we have presented EvoGraph, an accelerator-based high-performance incremental graph processing framework for processing time-evolving graphs. Using its novel programming model I-GAS that is based on Gather- Apply-Scatter model, EvoGraph computes graph properties only for the inconsistent subgraphs. Further with a user-tunable property-based optimization called property-guard for switching between incremental and static recomputation and deep memory copy operations via separate CUDA Streams, EvoGraph achieves improved system throughput when compared to competing streaming graph processing frameworks.

In summary, this dissertation provides hard evidence that new system design principles are required for efficient scheduling and fine grain resource management in heterogeneous many-core system, and that such methods are needed to harness the full potential of future exascale machines with compute nodes expected to be comprised of 1000s of accelerator and general purpose cores for the increasingly performance hungry applications with varied memory access pattern.

As an ongoing effort, we are exploring several extensions to the work in this dissertation. On the GPU sharing front, we are considering dynamic opportunities and trade-offs in mapping executions to either GPUs or CPUs, using runtime methods for dynamic binary translation. Another interesting extension that requires consideration would be further

exploration of the effects of data movement on program performance. This would consequently change scheduling policies for discrete vs. integrated GPUs. From the graph analytics viewpoint, we are investigating extensions of our solutions to support multiple on-node GPUs, going beyond single node processing to multi-node clusters and extreme-scale datasets. Specifically, understanding different kinds of consistency guarantees that are required to enable processing of evolving graph in a distributed setup. These new directions further enhance and extend the importance of GPU processing for scalable and time-sensitive data analytics problems.

REFERENCES

[1] "Adobe Photoshop.com: http://www.photoshop.com/,"

[2] "Amazon EC2 GPU cluster: http://aws.amazon.com/about-aws/whats-new/2010/11/15/announcing-cluster-gpu-instances-for-amazon-ec2/,"

[3] "Cage15: www.cise.ufl.edu/research/sparse/matrices/vanheukelum,"

[4] "CUDA 7.0: https://developer.nvidia.com/cuda-downloads/,"

[5] "Design document SSJ workload, SPECpower_ssj2008: http://www.spec.org/power_ssj2008,"

[6] "DIMACS10 ak2010: www.cise.ufl.edu/research/sparse/matrices/dimacs10,"

[7] "DIMACS10 belgium_osm: www.cise.ufl.edu/research/sparse/matrices/dimacs10,"

[8] "DIMACS10 coAuthorsDBLP: www.cc.gatech.edu/dimacs10/data/coauthor/,"

[9] "Elemental Technologies: http://tinyurl.com/bcj3jet,"

[10] "Modern GPU library:http://nvlabs.github.io/moderngpu/,"

[11] "Nimbix: http://www.nimbix.net/cloud-supercomputing/,"

[12] "NVIDIA cloud gaming: http://www.nvidia.com/object/cloud- gaming.html,"

[13] "NVIDIA Kepler GK110 white paper. 2012 https://www.nvidia.com/content/pdf/kepler/nvidia-kepler-gk110-architecture-whitepaper.pdf,"

[14] "OPENCV: http://opencv.org/,"

[15] "Peer1 hosting: http://www.peer1hosting.co.uk/hosting/gpu-servers,"

[16] "Penguin computing: http://www.penguincomputing.com/services/hpc-cloud/,"

[17] "Rodinia: https://www.cs.virginia.edu/ skadron/wiki/rodinia/index.php/rodinia:accelerating_compute-intensive_applications_with_accelerators,"

[18] "Top 500 List: http://www.top500.org/system/177975,"

[19] "Twitter Statistics: http://tinyurl.com/kcuhdcw,"

[20] "uk-2002: http://law.di.unimi.it/webdata/uk-2002/,"

[21] "Unified Virtual Addressing: http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/,"

[22] "University of Florida Sparse Matrix Collection http://tinyurl.com/hh8g3n9,"

[23] "VertexAPI2: http://www.royal-caliber.com/main.html. 2015,"

[24] "Wong, T., Preissl, R., Datta, P., Flickner, M., Singh, R., Esser, S., McQuinn, E., Appuswamy, R., Risk, W., and Simon, H., 10^{14} IBM Research Divsion, Research Report RJ10502, 2012,"

[25] "Yahoo WebScope: G7 webscope.sandbox.yahoo.com/catalog.php?datatype=g,"

[26] "Zillians: http://www.zillians.com/solutions/,"

[27] Amir, A., Datta, P., Risk, W. P., Cassidy, A. S., Kusnitz, J. A., Esser, S. K., Andreopoulos, E., Wong, T. M., Flickner, M., Alvarez-icaza, R., Mcquinn, E., Shaw, B., Pass, N., and Modha, D. S., "Cognitive computing programming paradigm: A corelet language for composing networks of neurosynaptic cores."

[28] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M., "Linkbench: A database benchmark based on the facebook social graph," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 1185–1196, ACM, 2013.

[29] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A., "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, pp. 187–198, Feb. 2011.

[30] Bahmani, B., Kumar, R., Mahdian, M., and Upfal, E., "Pagerank on an evolving graph," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, (New York, NY, USA), pp. 24–32, ACM, 2012.

[31] Becchi, M., Byna, S., Cadambi, S., and Chakradhar, S., "Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, (New York, NY, USA), pp. 82–91, ACM, 2010.

[32] Becchi, M., Sajjapongse, K., Graves, I., Procter, A., Ravi, V., and Chakradhar, S., "A virtual memory based runtime to support multi-tenancy in clusters with gpus," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, (New York, NY, USA), pp. 97–108, ACM, 2012.

[33] Bell, N. and Garland, M., "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[34] Blagodurov, S., Zhuravlev, S., Dashti, M., and Fedorova, A., "A case for numa-aware contention management on multicore systems," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2011.

[35] Brightwell, R., Barrett, B. W., Hemmert, K. S., and Underwood, K. D., "Challenges for high-performance networking for exascale computing," in *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pp. 1–6, Aug 2010.

[36] Broekema, P. C., van Nieuwpoort, R. V., and Bal, H. E., "Exascale high performance computing in the square kilometer array," in *Proceedings of the 2012 Workshop on High-Performance Computing for Astronomy Date*, Astro-HPC '12, (New York, NY, USA), pp. 9–16, ACM, 2012.

[37] Cadambi, S., Coviello, G., Li, C.-H., Phull, R., Rao, K., Sankaradass, M., and Chakradhar, S., "Cosmic: Middleware for high performance and reliable multiprocessing on xeon phi coprocessors," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, (New York, NY, USA), pp. 215–226, ACM, 2013.

[38] Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., and Snir, M., "Toward exascale resilience," *Int. J. High Perform. Comput. Appl.*, vol. 23, pp. 374–388, Nov. 2009.

[39] Cassidy, A. S., Merolla, P., Arthur, J. V., Esser, S. K., Jackson, B., Alvarez-icaza, R., Datta, P., Sawada, J., Wong, T. M., Feldman, V., Amir, A., dayan Rubin, D. B., Mcquinn, E., Risk, W. P., and Modha, D. S., "Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores," in *in International Joint Conference on Neural Networks (IJCNN). IEEE*, 2013.

[40] Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., and Muthukrishnan, S., "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, pp. 1804–1815, Aug. 2015.

[41] De Bosschere, K., D'Hollander, E., Joubert, G. R., Padua, D., Peters, F., and Sawyer, M., eds., *Applications, tools and techniques on the road to exascale computing*, vol. 22 of *Advances in Parallel Computing*. IOS press, Inc., 2012.

[42] Diamos, G. F., Kerr, A. R., Yalamanchili, S., and Clark, N., "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 353–364, ACM, 2010.

[43] Djidjev, H., Thulasidasan, S., Chapuis, G., Andonov, R., and Lavenier, D., "Efficient multi-gpu computation of all-pairs shortest paths," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 360–369, May 2014.

[44] Duato, J., Pena, A., Silla, F., Mayo, R., and Quintana-Ort, E., "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pp. 224–231, June 2010.

[45] Duong, N. T., Nguyen, Q. A. P., Nguyen, A. T., and Nguyen, H.-D., "Parallel pagerank computation using gpus," in *Proceedings of the Third Symposium on Information and Communication Technology*, SoICT '12, (New York, NY, USA), pp. 223–230, ACM, 2012.

[46] Ediger, D., Jiang, K., Riedy, J., and Bader, D., "Massive streaming data analytics: A case study with clustering coefficients," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–8, April 2010.

[47] Ediger, D., McColl, R., Riedy, J., and Bader, D., "Stinger: High performance data structure for streaming graphs," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pp. 1–5, Sept 2012.

[48] Ediger, D., Riedy, J., Bader, D., and Meyerhenke, H., "Tracking structure of streaming social networks," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1691–1699, May 2011.

[49] Esser, S. K., Andreopoulos, E., Datta, P., Barch, D., Amir, A., Arthur, J., Cassidy, A., Flickner, M., Merolla, P., Ch, S., Basilico, N., Carpin, S., Zimmerman, T., Zee, F., Alvarez-icaza, R., Kusnitz, J. A., Wong, T. M., Risk, W. P., Mcquinn, E., Nayak, T. K., Singh, R., and Modha, D. S., "Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores."

[50] Fard, A., Abdolrashidi, A., Ramaswamy, L., and Miller, J., "Towards efficient query processing on massive time-evolving graphs," in *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on*, pp. 567–574, Oct 2012.

[51] Farooqui, N., Schwan, K., and Yalamanchili, S., "Efficient instrumentation of gpgpu applications using information flow analysis and symbolic execution," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, (New York, NY, USA), pp. 19:19–19:27, ACM, 2014.

[52] Fu, Z., Personick, M., and Thompson, B., "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, (New York, NY, USA), pp. 2:1–2:6, ACM, 2014.

[53] Gharaibeh, A., Beltrão Costa, L., Santos-Neto, E., and Ripeanu, M., "A yoke of oxen and a thousand chickens for heavy lifting graph processing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 345–354, ACM, 2012.

[54] Gioiosa, R., "Towards sustainable exascale computing," in *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*, pp. 270–275, Sept 2010.

[55] Giunta, G., Montella, R., Agrillo, G., and Coviello, G., "A gpgpu transparent virtualization component for high performance computing clouds," in *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, EuroPar'10, (Berlin, Heidelberg), pp. 379–391, Springer-Verlag, 2010.

[56] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C., "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as*

part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), (Hollywood, CA), pp. 17–30, USENIX, 2012.

[57] Guevara, M., Gregg, C., Hazelwood, K., and Skadron, K., "Enabling task parallelism in the cuda scheduler."

[58] Guha, S., Cheng, B., and Francis, P., "Challenges in measuring online advertising systems," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, (New York, NY, USA), pp. 81–87, ACM, 2010.

[59] Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., and Ranganathan, P., "Gvim: Gpu-accelerated virtual machines," in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, (New York, NY, USA), pp. 17–24, ACM, 2009.

[60] Gupta, V., Schwan, K., Tolia, N., Talwar, V., and Ranganathan, P., "Pegasus: Coordinated scheduling for virtualized accelerator-based systems," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2011.

[61] Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., and Chen, E., "Chronos: A graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 1:1–1:14, ACM, 2014.

[62] Hong, S., Chafi, H., Sedlar, E., and Olukotun, K., "Green-marl: A dsl for easy and efficient graph analysis," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 349–362, ACM, 2012.

[63] Hong, S., Kim, S. K., Oguntebi, T., and Olukotun, K., "Accelerating cuda graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, (New York, NY, USA), pp. 267–276, ACM, 2011.

[64] Hong, S., Oguntebi, T., and Olukotun, K., "Efficient parallel graph exploration on multi-core cpu and gpu," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 78–88, Oct 2011.

[65] Jain, R., Chiu, D., and Hawe, W., "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," *CoRR*, vol. cs.NI/9809099, 1998.

[66] Kato, S., McThrow, M., Maltzahn, C., and Brandt, S., "Gdev: First-class gpu resource management in the operating system," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), pp. 37–37, USENIX Association, 2012.

[67] Khorasani, F., Vora, K., Gupta, R., and Bhuyan, L. N., "Cusha: Vertex-centric graph processing on gpus," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, (New York, NY, USA), pp. 239–252, ACM, 2014.

[68] Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A. D., Kaldewey, T., Lee, V. W., Brandt, S. A., and Dubey, P., "Fast: Fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 339–350, ACM, 2010.

[69] Kim, Y., Han, D., Mutlu, O., and Harchol-Balter, M., "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, Jan 2010.

[70] Krieder, S. J., Wozniak, J. M., Armstrong, T., Wilde, M., Katz, D. S., Grimmer, B., Foster, I. T., and Raicu, I., "Design and evaluation of the gemtc framework for gpu-enabled many-task computing," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, (New York, NY, USA), pp. 153–164, ACM, 2014.

[71] Kyrola, A., Blelloch, G., and Guestrin, C., "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 31–46, USENIX Association, 2012.

[72] Kysenko, V., Rupp, K., Marchenko, O., Selberherr, S., and Anisimov, A., "Natural language processing and information systems: 17th international conference on applications of natural language to information systems, nldb 2012, groningen, the netherlands, june 26-28, 2012. proceedings," (Berlin, Heidelberg), pp. 158–163, Springer Berlin Heidelberg, 2012.

[73] Lauderdale, C. and Khan, R., "Towards a codelet-based runtime for exascale computing: Position paper," in *Proceedings of the 2Nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '12, (New York, NY, USA), pp. 21–26, ACM, 2012.

[74] Leskovec, J., Lang, K. J., Dasgupta, A., and Mahoney, M. W., "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," 2008.

[75] Li, C., Song, S. L., Dai, H., Sidelnik, A., Hari, S. K. S., and Zhou, H., "Locality-driven dynamic gpu cache bypassing," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, (New York, NY, USA), pp. 67–77, ACM, 2015.

[76] Liedtke, J., "On micro-kernel construction," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, (New York, NY, USA), pp. 237–250, ACM, 1995.

[77] Low, Y., Gonzalez, J. E., Kyrola, A., Bickson, D., Guestrin, C. E., and Hellerstein, J., "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[78] Luk, C.-K., Hong, S., and Kim, H., "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 45–55, ACM, 2009.

[79] LUMSDAINE, A., GREGOR, D., HENDRICKSON, B., and BERRY, J., "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[80] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G., "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[81] McColl, R., Green, O., and Bader, D. A., "A new parallel algorithm for connected components in dynamic graphs," in *20th Annual International Conference on High Performance Computing*, pp. 246–255, Dec 2013.

[82] Merrill, D., Garland, M., and Grimshaw, A., "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 117–128, ACM, 2012.

[83] Merritt, A. M., Gupta, V., Verma, A., Gavrilovska, A., and Schwan, K., "Shadowfax: Scaling in heterogeneous cluster systems via gpgpu assemblies," in *Proceedings of the 5th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '11, (New York, NY, USA), pp. 3–10, ACM, 2011.

[84] Murphy, R. C., Wheeler, K., Barrett, B., and Ang, J. A., "Introducing the graph 500.," in *Cray Users Group (CUG)*, 2010.

[85] Phull, R., Li, C.-H., Rao, K., Cadambi, H., and Chakradhar, S., "Interference-driven resource management for gpu-based heterogeneous clusters," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, (New York, NY, USA), pp. 109–120, ACM, 2012.

[86] Preissl, R., Wong, T. M., Datta, P., Flickner, M., Singh, R., Esser, S. K., Risk, W. P., Simon, H. D., and Modha, D. S., "Compass: A scalable simulator for an architecture for cognitive computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 54:1–54:11, IEEE Computer Society Press, 2012.

[87] Rafique, M. M., Cadambi, S., Rao, K., Butt, A. R., and Chakradhar, S., "Symphony: A scheduler for client-server applications on coprocessor-based heterogeneous clusters," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, (Washington, DC, USA), pp. 353–362, IEEE Computer Society, 2011.

[88] Ramalingam, G. and Reps, T., "An incremental algorithm for a generalization of the shortest-path problem," *J. Algorithms*, vol. 21, pp. 267–305, Sept. 1996.

[89] Ravi, V. T., Becchi, M., Agrawal, G., and Chakradhar, S., "Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework,"

in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, (New York, NY, USA), pp. 217–228, ACM, 2011.

[90] Ravi, V. T., Ma, W., Chiu, D., and Agrawal, G., "Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, (New York, NY, USA), pp. 137–146, ACM, 2010.

[91] Righter, R. and Shanthikumar, J. G., "Scheduling multiclass single server queueing systems to stochastically maximize the number of successful departures," *Probability in the Engineering and Informational Sciences*, vol. 3, pp. 323–333, 7 1989.

[92] Rossi, R. A. and Ahmed, N. K., "Networkrepository: A graph data repository with visual interactive analytics," vol. abs/1410.3560, 2014.

[93] Roy, A., Mihailovic, I., and Zwaenepoel, W., "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 472–488, ACM, 2013.

[94] Schenk, O., Wchter, A., and Weiser, M., "Inertia-revealing preconditioning for large-scale nonconvex constrained optimization," *SIAM Journal on Scientific Computing*, vol. 31, no. 2, pp. 939–960, 2009.

[95] Sengupta, D., Agarwal, K., Song, S. L., and Schwan, K., "Graphreduce: Large-scale graph analytics on accelerator-based hpc systems," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pp. 604–609, May 2015.

[96] Sengupta, D., Belapure, R., and Schwan, K., "Multi-tenancy on gpgpu-based servers," in *Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '13, (New York, NY, USA), pp. 3–10, ACM, 2013.

[97] Sengupta, D., Goswami, A., Schwan, K., and Pallavi, K., "Scheduling multi-tenant cloud workloads on accelerator-based systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 513–524, IEEE Press, 2014.

[98] Sengupta, D., Song, S. L., Agarwal, K., and Schwan, K., "Graphreduce: Processing large-scale graphs on accelerator-based systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 28:1–28:12, ACM, 2015.

[99] Sengupta, D., Sundaram, N., Zhu, X., Willke, T. L., Young, J., Wolf, M., and Schwan, K., "Graphin: An online high performance incremental graph processing framework," in *Proceedings of the 22nd International European Conference on Parallel Processing*, Euro-Par '16, (Grenoble, France), 2016.

[100] Sengupta, D., Wang, Q., Volos, H., Cherkasova, L., Li, J., Magalhaes, G., and Schwan, K., "A framework for emulating non-volatile memory systemswith

different performance characteristics," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, (New York, NY, USA), pp. 317–320, ACM, 2015.

[101] Shalf, J., Dosanjh, S., and Morrison, J., "Exascale computing technology challenges," in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, (Berlin, Heidelberg), pp. 1–25, Springer-Verlag, 2011.

[102] Shi, L., Chen, H., Sun, J., and Li, K., "vcuda: Gpu-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, pp. 804–816, June 2012.

[103] Shun, J. and Blelloch, G. E., "Ligra: A lightweight graph processing framework for shared memory," pp. 135–146, 2013.

[104] Silva, M., Hines, M. R., Gallo, D., Liu, Q., Ryu, K. D., and Silva, D. d., "Cloudbench: Experiment automation for cloud environments," in *Proceedings of the 2013 IEEE International Conference on Cloud Engineering*, IC2E '13, (Washington, DC, USA), pp. 302–311, IEEE Computer Society, 2013.

[105] Silva, N. B., Tsang, I. R., Cavalcanti, G. D. C., and Tsang, I. J., "A graph-based friend recommendation system using genetic algorithm," in *IEEE Congress on Evolutionary Computation*, pp. 1–7, July 2010.

[106] Snavely, A. and Tullsen, D. M., "Symbiotic jobscheduling for a simultaneous multithreaded processor," *SIGARCH Comput. Archit. News*, vol. 28, pp. 234–244, Nov. 2000.

[107] Song, S., Su, C., Rountree, B., and Cameron, K. W., "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 673–686, May 2013.

[108] Song, S. and Cameron, K., "System-level power-performance efficiency modeling for emergent gpu architectures," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 473–474, ACM, 2012.

[109] Sun, J., Faloutsos, C., Papadimitriou, S., and Yu, P. S., "Graphscope: Parameter-free mining of large time-evolving graphs," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, (New York, NY, USA), pp. 687–696, ACM, 2007.

[110] Tarditi, D., Puri, S., and Oglesby, J., "Accelerator: Using data parallelism to program gpus for general-purpose uses," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 325–335, ACM, 2006.

[111] Teodoro, G., Sachetto, R., Sertel, O., Gurcan, M. N., Meira, W., Catalyurek, U., and Ferreira, R., "Coordinating the use of gpu and cpu for improving performance of compute intensive applications," in *Cluster Computing and*

Workshops, 2009. CLUSTER '09. IEEE International Conference on, pp. 1–10, Aug 2009.

[112] Valiant, L. G., "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, Aug. 1990.

[113] Vallis, O., Hochenbaum, J., and Kejariwal, A., "A novel technique for long-term anomaly detection in the cloud," in *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'14, (Berkeley, CA, USA), pp. 15–15, USENIX Association, 2014.

[114] Vora, K., Koduru, S. C., and Gupta, R., "Aspire: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, (New York, NY, USA), pp. 861–878, ACM, 2014.

[115] Wang, L., Huang, M., and El-Ghazawi, T., "Towards efficient gpu sharing on multicore processors," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 119–124, Oct. 2012.

[116] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J., "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, (New York, NY, USA), pp. 38:1–38:12, ACM, 2007.

[117] Yang, J. and Leskovec, J., "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, MDS '12, (New York, NY, USA), pp. 3:1–3:8, ACM, 2012.

[118] Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., and Catalyurek, U., "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, (Washington, DC, USA), pp. 25–, IEEE Computer Society, 2005.

[119] Zhong, J. and He, B., "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 1543–1552, June 2014.

VITA

Dipanjan Sengupta was born to Mr. Indrajit Sengupta and Mrs. Ranjana Sengupta in Kolkata, a city in West Bengal India. He graduated with a Bachelor of Technology in Computer Science and Engineering from Indian Institute of Technology Kharagpur (IIT KGP) in May 2008. After his Bachelors, he worked in Adobe systems as a part of Photoshop Elements team (PSE) for 3 years. He is a PhD candidate at the College of Computing, Georgia Institute of Technology, Atlanta, USA advised by Prof. Karsten Schwan and Dr. Matthew Wolf. His research at Georgia Tech focuses on scheduling and resource management in high performance, many-core accelerator-based systems.